Preface

The term application in this context is used to connote the kind of software for use by general public without any knowledge of the computing device for utilizing the software. On the other hand, software developers use system programs such as compilers and are generally aware of the system calls to the operating system for which they are building an application.

Small computing devices continue to surface while desktop and laptop operating systems are frozen among a few, some of which have long past their retirement. Small, in particular handheld computing devices and cell-phones enter the market loaded with a few standard applications such as email clients, or offer specific services.

None of the underlying operating systems are, or should be distributed. Consequently it is not possible to create a distributed language for developing diverse distributed applications appropriate for the next age of computing. Due to the lack of a distributed language we have extended the era of client-server model as the only form of distributed computing by keeping our software applications stationary.

It is practically impossible to offer a new operating system for desktop devices. The number of applications available for incumbent operating systems is too large to pack with a new operating system. In addition, users are reluctant to switch from a familiar look and feel to a whole new presentation of the same application. Thus, applications for a new operating system must preserve much of the look of standard applications.

Software companies are effectively left out of developing applications for handheld computing devices. Even with regard to desktop applications software companies generally specialize in one operating system. Occasionally third-party libraries allow software companies to offer their applications for more than one operating system.

Instructors cannot learn all the languages and technologies in order to teach software engineering. Inevitably they specialize in a specific language on a particular computing device. A set of ad hoc technologies for developing software on a specific computing device is usually referred to as a platform. A substantial part of the knowledge needed for developing software in a platform is specific to that platform and not transferable to other platforms. This makes it difficult to identify software engineering properly. The process of development is heavily tied to the platform.

While libraries have a variety of uses, increasing expressiveness is not one of them. Increasing the size of library for a weak language does no more than turning the language into an interface to the library. Effectively the language becomes an interpreter for using the library as BASIC is to its tokens, except the engineer becomes the interpreter for selecting library calls.

Indeed, if enlarging the library would increase the expressiveness of a language there would be no need for C++, or any other language beyond C, for that matter. Even though C++ translates to C, yet the C compiler cannot make sense of private members, for instance. Therefore, any size of library added to C will not enable C compiler to make sense of the notion of privacy of members and methods.

To solve the problems mentioned above we need a universal computing language that remains the same independent of the computing device. This will allow producing software engineers who are able to solve problems of automation within an abstract language, just like mathematics is used in other forms of engineering. In other words, for developing applications (as opposed to system programs) we need an abstract programming language, available uniformly for all computing devices.

Unlike mathematics a computing language, however abstract, requires an engine. A universal language for developing software independent of computing devices must rest on an engine available on all computing devices. Furthermore, this engine must be self-contained in supporting such abstractions as processes, threads, signaling etc. For resources such as graphics and files it can simply act as a pass-through to the underlying operating system.

Thus, the abstract language ultimately needs the services of an operating system, as opposed to a virtual machine. So this problem must be solved from bottom up. First, we must create a universal distributed operating system that can coexist with the operating system that controls the resources of a computing device. Only then, we will be able to design an abstract distributed language for developing applications that would be available on all computing devices.

The abstract language cannot be a minimal language like C, which is intended for system programming. Software development with a fat and yet minimal language, such as Java is more painful than using a system programming language like C. Engineers already know that even the facilities of C++ are inadequate for developing contemporary applications. Therefore the intended abstract language must provide sufficient linguistic abstractions for dealing with complexities of contemporary applications without the need for being aware of system calls specific to any operating system.

Once an abstract computing language is provided for developing abstract software, component-oriented development could make sophisticated components available independent of computing devices. System specific computational statements, such as graphical user interface can also be cast on universal tools. Thus, a software developer should be almost as oblivious of the computing device for which he or she is developing an application, as the user of that application.

Z47 Processor is a distributed operating system that can coexist with any operating system in control of a computing device. Among other things, Z47 manages its own threads, processes, signals and inter-process communications. Otherwise put, Z47 is self-contained.

The abstract language Z++ relies on Z47. Z++ compiler maps the linguistic abstractions of Z++ to system calls to Z47. For specialized services, such as touch-screen and graphical user interface, Z47 makes system calls to the underlying operating system. Nonetheless, Z++ remains abstract, even though some services will not be available for some computing devices (which the compiler will inform the engineer). After all, unless a computing device supports touch-screen no language will be able to provide that kind of capability for applications targeted for that device.

The Z++ compiler and the Z47 Processor are both written in C++, and ultimately in C. Nonetheless, what the sum of Z47 and Z++ can do, C and C++ cannot. For instance, autonomous agents with full (strong) mobility are out of range for any language other than Z++. In addition, expressiveness of Z++ for managing complex contemporary applications exceeds any other language, by far.

Introduction

Geometry and later algebra began their leap only after their fundamental notions were divested from physical objects that represented those notions. These abstract notions dissociated from their physical representations paved the way for studying geometry and algebra independent of other physical properties associated with the representations of their objects of study.

The process of abstraction, mentioned above, is only useful when its findings can be applied to the physical world through generalization. Otherwise, it is no more useful than a game like chess. A special form of generalization, called modeling, is particularly important. Like mechanics or electricity, it is the ultimate desire of scientists to find a mathematical model for their science.

The process of automation with the help of a programming language began its speedy evolution after a few programming notions were divested from hardware circuitry. Since ultimately a programming language relies on hardware for the execution of its statements, the path of finding linguistic abstraction has been difficult. The apparent resemblance of statements of a programming language with mathematics has impeded progress by identifying constituents, like functional paradigm, as the whole approach to designing a programming language.

The notion of abstract data types, and later object-oriented programming were two major steps next to structural statements of selection and iteration. Specifically, the evolution of the notion of class by endowing it with invariants, constraints, threading and templates has provided a means for modeling conceptual objects, or physical objects by way of abstraction. The notion of class for automation is similar to the notion of abstract space in mathematics.

The notion of class is not the end of evolution. Many physical or conceptual entities that we wish to automate consists of larger interacting entities. For instance, a business consists of several departments each of which may consist of several sub departments, etc. The automation of such a complex entity is recursive in nature. This directs us towards the notion of module or component. A module is a program made up of classes and their interactions. The important property of a module is that one can build larger modules by bringing together several modules.

The notion of a traveling module (autonomous agent) is an abstraction of automation unrelated to the notion of class. A module is a program, which maps to the notion of process. Since a process is an entity of an operating system, the realization of the programming notion of traveling module requires separating the various roles of an operating system. The Virtual Processor Z47 is a distributed operating system, which makes the realization of autonomous agents as a programming concept, possible.

This book explains and illustrates the programming language Z++, powered by Z47. Z++ is not an extension of C++. However, the elementary parts of Z++ language use the same syntax as C++ for continuity of knowledge from one language to the other. Even the small part of Z++ that resembles C++ has been enhanced syntactically, and in many cases semantically.

C++ is a rudimentary language with its only attractions borrowed from the two languages Simula and C. Although there is a need for an object-oriented system programming language like C++, the weak implementation of its abstractions will have to be replaced so the language can stand up to future programming demands.

This book explains all aspects of Z++, and illustrates all major points with actual examples with output. All Z++ libraries are explained. While it is not meant for self-study, it is the standard for writing such books, and for professionals. A C++ engineer can use this book for self-study.

Table of Contents

Chapter I. Fundamental Types

Chapter II. Control Structures
- Chapter II.1. Selection Structures
  - Chapter II.1.A. The if-elsif-else-endif statement
  - Chapter II.1.B. The switch-endswitch statement
    - Section II.1.C.I. Range and Sequence for case values
  - Chapter II.1.C. The conditional statement
- Chapter II.2. Iteration Structures
  - Chapter II.2.A. The for-endfor statement
  - Chapter II.2.B. The while-endwhile statement
  - Chapter II.2.C. The do-enddo statement
- Chapter II.3. Atomic (critical) Section

Chapter III. Type Definition Mechanisms
- Chapter III.1. Enumeration
  - Section III.1.A. Privacy and Resolution of Literals
  - Section III.1.B. Successor and Predecessor
  - Section III.1.C. Enumeration Bracket Operator
  - Section III.1.D. Extending an Enumeration Type
- Chapter III.2. Class
  - Section III.2.A. Member Visibility
  - Section III.2.B. Default Methods
  - Section III.2.C. Default Copy Constructor
  - Section III.2.D. Default Assignment Operator
  - Section III.2.E. Calls on Arrays of Objects
  - Section III.2.F. Array Constructions
  - Section III.2.G. Dynamic Array Members
  - Section III.2.H. Inheritance
  - Section III.2.I. Class Invariants
  - Section III.2.J. Method Constraints
  - Section III.2.K. Common Members
  - Section III.2.L. Friendship
  - Section III.2.M. Explicit Cast Specification
  - Section III.2.N. Const Specification
  - Section III.2.O. Polymorphism
  - Section III.2.P. Inheritance and Casting
- Chapter III.3. Task

Section III.3.A. Task Idler
Section III.3.B. Task Signal Handler
Section III.3.C. Task Accept Signals
Section III.3.D. Task Hear Signals
Section III.3.E. Arrays of Tasks
Section III.3.F. Task Templates
Section III.3.G. Exceptions in Tasks

Chapter III.4. Components and Modules
Section III.4.A. Local Modules
Section III.4.B. Loading and unloading Modules
Section III.4.C. Module Context
Section III.4.D. Types of arguments and return object
Section III.4.E. Module and Invariants
Section III.4.F. External methods and Constraints
Section III.4.G. Module Inheritance
Section III.4.H. Remote Linkage
Section III.4.H.1. Local loading of a Remote Component
Section III.4.H.2. Remote loading of a Remote Component
Section III.4.I. Threads and Components

Chapter III.5. Linking with other languages
Section III.5.A. Static Linkage Modules
Section III.5.A.1. Reuse through Z++ Modules

Section III.5.B. Dynamic Linkage Modules
Section III.5.B.1. Example for Dynamic Linkage Modules
Section III.5.B.2 Sample of a generated source

Section III.5.C. Linkage Module Options
Section III.5.D. A namespace example
Section III.5.E. Linkage Data Types
Section III.5.F. Remote Linkage and RPC
Section III.5.G. Linkage Module and SOA

Chapter III.6. Union
Section III.6.A. Assignment and union members

Chapter III.7. Bitfields

Chapter III.8. Collection
Section III.8.A. Defining a Collection
Section III.8.B. Collection Tag
Section III.8.C. Shared Methods
Section III.8.D. Shared Method Inheritance
Section III.8.E. Extending Collections

Section III.8.F. Invoking a Shared Method
Section III.8.G. Protecting tag and values
Section III.8.H. Collection Templates
Section III.8.I. Polymorphism and Casting

Chapter III.9. Typedef

Chapter III.10. Type and Function pointer

Chapter III.11. Pointer
Section III.11.A. Pointer Casting
Section III.11.B. Pointer Dereferencing

Chapter III.12. Array
Section III.12.A. Static and Dynamic Arrays
Section III.12.B. Degenerate Arrays
Section III.12.C. Sizes of dimensions of Dynamic arrays
Section III.12.D. Array support for fundamental types
Section III.12.E. Arrays and Overloading operators
Section III.12.F. Array Conversions
Section III.12.G. Pointer Syntax for Arrays

Chapter III.13. Frame and Canvas

Chapter IV. Abstract Data Types (Templates)
Chapter IV.1. Template Definition and Instantiation
Chapter IV.2. Template Parameter Specification (Pattern)
Chapter IV.3. Template Typedef
Chapter IV.4. Task as Template Instantiation
Chapter IV.5. Template Namespace

Chapter V. Namespace
Chapter V.1. Namespace Sections
Chapter V.2. Namespace Implementation
Chapter V.3. Namespace Derivation
Chapter V.4. Protected Namespace
Chapter V.5. Scope of a Namespace
Chapter V.6. Namespace Example
Chapter V.7. Template Namespace Implementation

Chapter VI. The notion of Signal
Chapter VI.1. Preliminary Definitions
Chapter VI.2. Generating and Catching signals
Chapter VI.3. Process-bounded Singular signals
Chapter VI.4. Node-bounded Singular signals
Chapter VI.5. Process-bounded Entire signals

Section VI.5.A. Registration for Global Threads
Section VI.5.B. Registration for Task Threads
Section VI.5.C. Task Handlers
Section VI.5.D. Example for Entire Signals

Chapter VI.6. Node-bounded Entire signals
Chapter VI.7. Extending Entire signals
Section VI.7.A. Example for extending Entire Signals

Chapter VII. Distributed Signaling
Chapter VII.1. Preliminaries
Chapter VII.2. Acceptance and Rejection of Tell signals
Chapter VII.3. Data transmitted with Tell signals
Chapter VII.4. Registering Hear signals
Chapter VII.5. Registration for Global Threads
Chapter VII.6. Registration for Task Threads
Chapter VII.7. Task Handlers
Chapter VII.8. Telling Distributed signals
Chapter VII.9. Further support for Tell-Hear signals
Chapter VII.10. Concurrent Communicating Processes
Chapter VII.11. Web Services
Chapter VII.12. Tell-Hear Example
Chapter VII.13. Task Tell-Hear Example

Chapter VIII. Exception Mechanism
Chapter VIII.1. Extending System Exceptions
Chapter VIII.2. Raising Exceptions
Chapter VIII.3. The structure of Layer Statement
Chapter VIII.4 Handling Exceptions
Chapter VIII.5 Else and Elsall
Chapter VIII.6. Resumption
Chapter VIII.7. Shallow and Deep Exceptions
Chapter VIII.8. Throws specification
Chapter VIII.9. Propagation of Exceptions
Chapter VIII.10. Branching of Exceptions

Chapter IX. Conversions
Chapter IX.1. Conversions for Fundamental types
Chapter IX.2. Conversions for Type Constructors
Chapter IX.3. Array Conversions
Chapter IX.4. Conversions for Symmetric operators
Chapter IX.5. Conversions for Asymmetric operators

Chapter X. Operators
Chapter X.1. Assignment Operators
Section X.1.A. Numeric Assignment

Section X.1.B. Bitfields Assignment
Section X.1.C. Function Assignment
Section X.1.D. Array Assignment
Section X.1.E. Special Assignments
Section X.1.F. Enumeration Assignments

Chapter X.2. Arithmetic Operators

Chapter X.3. Comparison Operators
Section X.3.A. Chaining Comparison Operators
Section X.3.B. Overloading Comparison Operators
Section X.3.C. String Comparisons
Section X.3.D. Conversions in Chains
Section X.3.E. Enumeration Comparisons
Section X.3.F. Pointer Comparisons
Section X.3.G. Array Comparisons

Chapter X.4. Logical Operators
Section X.4.A. Negation of Pointers and Arrays
Section X.4.B. Overloading Logical Operators
Section X.4.C. Operand Type for Logical expressions

Chapter X.5. Increment/Decrement Operators
Section X.5.A. Inc/Dec of Pointers
Section X.5.B. Inc/Dec of Enumerations
Section X.5.C. Inc/Dec of Arrays
Section X.5.D. Inc/Dec of Bitfields

Chapter X.6. Unary Prefix Operators
Section X.6.A. Address operator &
Section X.6.B. De-reference operator *
Section X.6.C. Bit Inversion operator ~
Section X.6.D. Minus operator -

Chapter X.7. Special Operators
Section X.7.A. Scope Operator
Section X.7.B. Derivation Operator
Section X.7.C. Structure Operators
Section X.7.D. Comma Operator
Section X.7.E. Conditional Expression Operators
Section X.7.F. Signaling Operators
Section X.7.G. Size Operator
Section X.7.H. Cast Operator
Section X.7.I. New Operator
Section X.7.J. Delete Operator
Section X.7.K. Destroy Operator

Section X.7.L. Conversion Operator

Chapter XI. Global Threads
Chapter XI.1. Inter-thread Communication
Chapter XI.2. Registration of Entire Signals
Chapter XI.3. Registration of Hear Signals
Chapter XI.4. Disengage Signals
Chapter XI.5. The use of Mutex

Chapter XII. Autonomous Agent (travel statement)
Chapter XII.1. Travel statement and entry points
Chapter XII.2. One travel statement per entry point
Chapter XII.3. States of global objects
Chapter XII.4. Types of objects transported
Chapter XII.5. Multi-threaded and Component Agents
Chapter XII.6. Exceptions raised by travel statement
Chapter XII.7. Autonomous Agent Examples
Section XII.7.A. Scopes and Datatypes
Section XII.7.B. Nested Exception Scopes
Section XII.7.C. Handling Agent Exceptions
Section XII.7.D. Handling Agent Exceptions – 2

Appendices
Appendix I. Structure of a Z++ program

Appendix II. Preprocessor Commands
Appendix II.1. The include command
Appendix II.2. The define command
Appendix II.3. The undef command
Appendix II.4. The if-else-endif structure
Appendix II.5. The ifnot-else-endif structure
Appendix II.6. The continuation character

Appendix III. Preprocessor Macro Command

Appendix IV. Standard Library
Appendix IV.1. Input and Output
Appendix IV.1.A. Output
Appendix IV.1.B. Input
Appendix IV.1.C. Overloading IO operators

Appendix IV.2. File Operations
Appendix IV.2.A. FileStreamType

Appendix IV.2.B. Opening a file
Appendix IV.2.B.1. Setting input/output direction

Appendix IV.2.B.2. Truncating file
Appendix IV.2.B.3. Appending to an existing file
Appendix IV.2.B.4. Setting binary/text file mode

Appendix IV.2.C. Input from a file
Appendix IV.2.C.1. Input from a text file
Appendix IV.2.C.2. Reading from a binary file

Appendix IV.2.D. Output to a file
Appendix IV.2.D.1. Output to a text file
Appendix IV.2.D.2. Writing to a binary file

Appendix IV.2.E. Moving around in a file
Appendix IV.2.E.1. Seek actions
Appendix IV.2.E.2. Setting seek origin

Appendix IV.2.F. Locating position of input/output

Appendix IV.2.G. Examples
Appendix IV.2.G.1. Text File
Appendix IV.2.G.2. Binary File
Appendix IV.2.G.3. File Read/Write

Appendix IV.3. Character Routines
Appendix IV.3.A. isSpace
Appendix IV.3.B. isDigit
Appendix IV.3.C. isAlpha
Appendix IV.3.D. isUpper
Appendix IV.3.E. isLower
Appendix IV.3.F. isSymbol
Appendix IV.3.G. isOperator
Appendix IV.3.H. isBracket
Appendix IV.3.I. toUpper
Appendix IV.3.J. toLower

Appendix IV.4. Memory Routines
Appendix IV.4.A. memset
Appendix IV.4.B. memcpy
Appendix IV.4.C. memcmp
Appendix IV.4.D. Example

Appendix IV.5. String Routines
Appendix IV.5.A. strstr
Appendix IV.5.B. strchr
Appendix IV.5.C. getchr
Appendix IV.5.D. Left

Appendix IV.5.E. Right
Appendix IV.5.F. Middle
Appendix IV.5.G. toUpper
Appendix IV.5.H. toLower
Appendix IV.5.I. Replace
Appendix IV.5.J. Reverse
Appendix IV.5.K. strcpy
Appendix IV.5.L. strncpy
Appendix IV.5.M. Examples

Appendix IV.6. Time Routines
Appendix IV.6.A. Example

Appendix IV.7. Socket Library
Appendix IV.7.A. Constructor
Appendix IV.7.B. Input
Appendix IV.7.C. Output
Appendix IV.7.D. Send
Appendix IV.7.E. Receive
Appendix IV.7.F. Example

Appendix IV.8. Client Server Model
Appendix IV.8.A. Server class
Appendix IV.8.B. Server Initialization
Appendix IV.8.C. Server accept wait-loop
Appendix IV.8.D. The Service Method
Appendix IV.8.E. Server Termination

Appendix IV.8.F. Client class
Appendix IV.8.G. Client Initialization
Appendix IV.8.H. Connecting to Server
Appendix IV.8.I. The Connection Socket
Appendix IV.8.J. Client Termination

Appendix IV.9. Database Interface
Appendix IV.9.A. Library Interface
Appendix IV.9.A.1. Connecting to a Database
Appendix IV.9.A.2. Establishing a Session
Appendix IV.9.A.3. Ending a database Session

Appendix IV.9.B. Language Interface
Appendix IV.9.B.1. SQL select statement
Appendix IV.9.B.2. SQL conditional clauses
Appendix IV.9.B.3. Z++ expressions in SQL conditions
Appendix IV.9.B.4. SQL insert statement

Appendix IV.9.B.5. SQL delete statement
Appendix IV.9.B.6. SQL update statement
Appendix IV.9.B.7. SQL fetch statement
Appendix IV.9.B.8. SQL free statement

Appendix IV.9.C. A complete example

Appendix IV.9.D. Proxy Database Server

Appendix IV.9.E. Database kind specific parameters
Appendix IV.9.E.1. MySQL Parameters
Appendix IV.9.E.2. Ingres Parameters
Appendix IV.9.E.3. Postgres Parameters

Appendix IV.9.F. Database Exceptions

Appendix.IV.9.G. Datatypes for columns

Appendix V. Z++ Template Library (ZTL)
Appendix V.A. List, Stack and Queue
Appendix V.A.1. List
Appendix V.A.2. Stack
Appendix V.A.3. Queue
Appendix V.A.4. Iterator
Appendix V.A.5. Example

Appendix V.B. Vector
Appendix V.C. Array
Appendix V.D. Hash Table
Appendix V.E. Heap

Appendix VI. Parameter Default Initializers

Chapter I. Fundamental Types

We need a few built-in (undefined) types in order to define more complex types. The Compiler knows the fundamental types and their properties. This allows us to instruct the Compiler to build more complex types via Type Definition Mechanisms.

The built-in types of Z++ are **numeric**, **string**, **boolean** and **mutex**.

Numeric types. Z++ numeric types are same as C numeric types, except **long** and **ulong** are 64 bits. The term **unsigned** is not used in Z++. Instead the type-name is prefixed with the character u, as follows: **uchar**, **ushort**, **uint** and **ulong**. The numeric types **float** and **double** are identical to those of C.

String type. The type **string** is built-in in Z++. As illustrated in Switch Statement, objects of type **string** can be used as **case** values for the **switch** control structure.

Objects of type **string** can be compared, assigned and concatenated. The **size** operator returns the length of a **string** object. Furthermore, one-dimensional **char** arrays and objects of **string** type can be assigned to one another. See also Special Assignments.

The following example illustrates some built-in **string** operations. The String Library provides a set of functions for further **string** manipulations.

```
// Samples.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      string first = "First";
      string second = "Second";

// Comparison

      if (first != second)
            output << "Strings are not equal\n";
      endif;

// Concatenations

      string both = first + " and " + second;
      output << both << '\n';
```

```
        first += second;
        output << first << '\n';

// Length of string

        output << "Length of string is: " << size(both) << '\n';

        output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Strings are not equal
First and Second
FirstSecond
Length of string is: 16
Good-bye World!
```

Boolean type. The type **boolean** is built-in. In particular, conditions must be of **boolean** type in that numeric objects cannot be tested for true or false. As a special case, a pointer type is directly tested for nullity, as explained in Pointer Nullity.

Mutex type. The type **mutex** is built-in. Operator ++ locks a **mutex** object, and -- unlocks it. Objects of type **mutex**, like instances of **task** type, cannot be compared or assigned. Similarly, **mutex** objects cannot be passed by value to function calls. Instead, use pass by reference, or pointers.

An object of type **mutex** is for mutual exclusion of threads attempting to **access** a **common** resource. A thread T attempting to lock a **mutex** object will face one of two situations. If the **mutex** object is already locked by another thread, T will be blocked and put in a waiting **queue**. Otherwise, T will succeed in locking the **mutex** object and continue its execution.

Threads blocked in the **waiting queue** of a **mutex** object become **unblocked** when the executing thread **unlocks** the **mutex** object.

For an example illustrating the use of **mutex** see The use of Mutex. For another form of controlling the execution of threads see Atomic Section.

Chapter II. Control Structures

Definitions, like defining a class, are processed by the compiler. The compiler may generate default code, for instance for the copy constructor. Nonetheless, definitions are considered none-executable with the implication that no corresponding executable code is generated.

On the other hand, the statements of the body of a function are executable. At load time the point of execution is set at an **entry** point, usually the **main entry** point. From that point on, due to threading, the program may obtain several points of execution. Each point of execution is called a control point, meaning the point in the program that has received control of the processor for execution of statements.

The **normal behavior** of all control points follows the same patterns of **sequence**, **selection** and **iteration**. Abnormal behaviors are discussed in Exception Mechanism. The pattern used most is sequence, where statements are executed one after the other. We use the term **block** (of code) when referring to a sequence that may contain simple statements like declarations and assignments, as well as selections and iterations.

The color of programming is the ability to select a block of code for execution, from a set of blocks based on run-time **conditions**. Iteration statements simplify repeated execution of a block of code.

In the following chapters we will illustrate the mechanisms for writing selection and iteration statements. Since these mechanism change the point of execution (control point), they are known as control structures. The term structure emphasizes the absence of the goto statement, which causes ad hoc jumping of the control (of execution).

For clarity, we mention our use of the term of **block** (of code) in this chapter. A block is a sequence of statements, with a **scope**. The statements in a block could be nested (recursively) selections, iterations and function calls. The **scope** property of a block indicates that within a block new objects can be created via declarative statements. This implies that a block has a starting and an ending point.

All control structures have a corresponding **closing tag**. Blocks making up a control structure are also called its **legs**. The end and start of each leg (as a block) is determined by the keywords of the control structure. In particular, the keyword **break** is not used as an ending tag of a block.

Chapter II.1. Selection Structures

The control structure selection puts color in programming. Without the ability to select different paths based on current conditions automation will have very limited application.

In Z++ there are three control structures that can be used in making selection statements. The **if-elsif-else-endif** uses a condition to select the block to execute next. A condition is a boolean expression involving the states of one or more objects. The **switch-endswitch** control structure uses the (literal) value of an object to select the next block for execution. The **conditional** control structure uses a condition but the statement as a whole evaluates to an object, which can be used in any expression, such as an arithmetic expression.

Chapter II.1.A. The if-elsif-else-endif statement

The simplest form of this selection control structure has a single leg as follows.

```
if (boolean expression)
// Block of code
endif;
```

After the evaluation of the condition, the block of code will be selected for execution **only if** the condition yields **true**. Otherwise, the control of execution will skip the block and **move** to the **first** statement following the closing tag **endif**.

The next form of this control structure has two legs, as shown below. In this case, exactly one of two legs will be selected for execution. The **else** leg will be selected **only if** the condition yields **false**.

```
if (boolean expression)
// Block of code
else
// Block of code
endif;
```

In many situations we deal with a number of conditions. Based on the truth of one of these conditions, we wish to select its block for execution. However, we want exactly one leg to be selected for execution. In particular, if all conditions evaluate to false, we want the **else** leg to be selected for execution.

It is important to know that, there is a possibility that several conditions may be true simultaneously. At execution time, the conditions are tested one after the other from top down. The first condition that yields true, its leg will be selected for execution.

In general, there can be one or more **elsif** legs, and the **else** leg is optional.

```
if (boolean expression)
// Block of code
elsif (boolean expression)
// Block of code
// more elsif legs …
elsif (boolean expression)
// Block of code
else
// Block of code
endif;
```

This is one single statement, and exactly one leg will be selected for execution. In the absence of the **else** leg, when all conditions yield false, the entire statement is skipped and the control of execution moves **directly** to the first statement after **endif**.

Suppose you have two conditions, and you wish to execute a block of code whenever each condition is true, or when both conditions are false you wish to execute another block of code. You can nest the **if-endif** as follows.

```
if (condition-1)
      // block-1
      if (condition-2)
            // block-2
      else
            // else block-2
      endif; // condition-2
elsif (condition-2)
      // block-2
      if (condition-1)
            // block-1
      else
            // else block-1
      endif; // condition-1
else
      // block of code when both conditions fail
endif;
```

Chapter II.1.B. The switch-endswitch statement

The **switch** control structure selects the block to execute based on the value of its operand. The types of operands for the **switch** statement are discrete numeric, from **char** to **long** and their corresponding unsigned ones from **uchar** to **ulong**. In addition, objects of **enumeration** and **string** type are also allowed.

Below is the basic form of a **switch** statement in Z++. The **operand** can be any **expression** that yields an object of acceptable type.

```
switch(operand)
      // initial block of code
      case value-1:
            // block of code
      case value-2:
            // block of code
// more cases ...
      else
            // block of code
endswitch;
```

The **else** leg is optional. The block selected for execution is the one whose **case** value is equal to the value of the **operand**. If the value of the operand does not match any of the **case** values, the **else** leg will be selected. In the latter case, in the absence of an **else** leg the entire **switch** statement is skipped, and control of execution moves **directly** to the first statement following the closing tag **endswitch**.

The objects declared in the initial block of code are available to all legs, including the **else** leg, and are destroyed at **endswitch**. In addition, each leg can declare its own local objects.

A **return** statement in the **initial block of code** can skip the **switch** statement. This is useful when certain conditions may apply.

Note that cases of a **switch** statement do not use **break**. A **case** leg with an empty block (without any statements) is called an empty leg. When an empty leg is selected for execution the control moves to the statement after **endswitch**. This will also skip the **else** leg, if there is one.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------ //

int fun(int n)
      return n += 2;
end;
```

```
// ------------------------------------------------------------------ //

enum secondEnum {_secondOne = 8, _secondTwo};

// firstEnum will include literals of secondEnum. _firstOne will be 10.

enum firstEnum : secondEnum {
      _firstOne, _firstTwo, _firstThree
};

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// switch operand can be an expression

      switch(fun(4))
            case 3: output << "3 selected\n";
            case 5: output << "5 selected\n";
            else output << "None selected\n"; //this is selected
      endswitch;

// strings as case values

      string s = "Tarzan";

      switch(s)
            case "Elephant": output << "Elephant\n";
            case "Tarzan": output << "Tarzan\n";
            case "Tiger": output << "Tiger\n";
      endswitch;

// Set firstObject to highest value: _firstThree.

      firstEnum firstObject = [[firstEnum]];

// The last leg is selected, printing 12.

      switch(firstObject)
            case _secondOne: output << [_secondOne] << '\n';
            case _secondTwo: output << [_secondTwo] << '\n';
            case _firstOne: output << [_firstOne] << '\n';
            case _firstTwo: output << [_firstTwo] << '\n';
            case _firstThree: output << [firstObject] << '\n';
      endswitch;

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
None selected
Tarzan
12
Good-bye World!
```

Section II.1.C.I. Range and Sequence for case values

In Z++ a sequence of **case** values cannot be written as case value-1: case value-2. This will make case value-1 an empty leg. Instead, several **case** values can be combined as follows, called a **sequence**.

```
case value-1, value-2, …, value-n:
```

If the values constitute a **range**, you can use the following.

```
case start-value .. end-value:
```

The range will include the `start-value` and the `end-value`. The two dots in between the values indicate a range. Note that the two dots together is a token and there is **no space** between them.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      char ch = 7;

// Range

      switch(ch)
            case 'a' .. 'z': output << "Lower case.\n";
            case 'A' .. 'Z': output << "Upper case.\n";
            case 0 .. 9: output << "Digit.\n";
            else output << "Not alphanumeric\n";
      endswitch;

// Sequence

      switch(ch)
            case 1,3,5,7,9: output << "Odd.\n";
            case 2,4,6,8: output << "Even.\n";
```

```
            case 0: output << "Zero.\n";
            else output << "Not a digit.\n";
      endswitch;

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Digit.
Odd.
Good-bye World!
```

Chapter II.1.C. The conditional statement

The control structure conditional is actually an expression with a final value that can be used in expressions in which the conditional statement appears. This control structure uses the symbols ? (question) and : (colon) and is explained in Conditional Expression.

Chapter II.2. Iteration Structures

Iteration control structures facilitate repeated execution of a block of code until a certain condition has been fulfilled. The **break** statement makes it possible to **exit** an iteration block, terminating repetition prior to the fulfillment of the iteration condition. On the other hand, based on specified conditions, the **continue** statement skips the execution of statements following it to the end of iteration block, for a **single** iteration.

In this chapter we discuss the iteration control structures of Z++. The **for-loop** generally uses counting for the fulfillment of the condition to terminate iteration. The **while-loop** tests the condition to terminate iteration **before** executing any of the statements of its iteration block. On the other hand, the **do-loop** tests the condition **after** executing the statements of its iteration block. Thus, the **while-loop** may iterate **zero** or more times, while the **do-loop** will iterate **one** or more times. The **for-loop** behaves like the **while-loop** and may not execute its block.

The body of iteration statements (block of code) can be **empty**.

Chapter II.2.A. The for-endfor statement

The general form of control structure for-loop as is as follows.

```
for (initialization ; condition ; tail computation)
      // Iteration block of code
endfor;
```

The initialization section is meant for declaring objects, and particularly for initializing the objects to be used for counting the number of iterations. Statements in this section must be separated with **comma operator**. Objects **declared** in the initialization section are **destroyed** when leaving the **for-loop** at **endfor**.

The condition section tests the condition, and will execute the iteration block only if the condition evaluates to **True**. Otherwise, the point of execution is **moved** to the statement following **endfor**. This section must be a **boolean** expression.

An **empty** condition for the **for-loop** is considered as True, which turns it into an infinite loop. This needs a **break** statement to terminate iteration.

The section for tail computation is executed when all the statements of the for-block have been executed. It is generally meant for incrementing counters, but other statements can appear there. Statements must be separated with **comma operator**.

The **continue** statement skips the execution of statements following it. This **moves** the execution control to the **tail** computation section. That is, the **continue** statement does **not** skip the tail computation section.

Chapter II.2.B. The while-endwhile statement

The syntax for **while** control structure has the following form.

```
while (boolean expression)
      // Block of code
endwhile;
```

The block of code will execute only if the **boolean** expression evaluates to True. Otherwise, the point of execution is **moved** to the statement following **endwhile**.

The statements **break** and **continue** can be used in a **while** loop, as well.

Chapter II.2.C. The do-enddo statement

The do-loop has two forms. The full form is as follows.

```
do
      // Block of code
enddo (boolean expression);
```

The condition is tested **after** the execution of the statements of the block of code. The iteration will continue so long as the **boolean** expression evaluates to **False**. That is, the loop **terminates** when the condition becomes **True**.

The following form of do-loop has **empty** condition, and is therefore an **infinite** loop. To terminate iteration you will need to use the **break** statement.

```
do
      // Block of code
enddo;
```

The above pattern is the **simplest** infinite loop in Z++. On the other hand, for **waiting** until a condition **becomes true**, a full do-loop with **empty** body provides a simple pattern. The condition is expected to change by factors not in control of the thread. For instance, some other thread may cause it to change.

```
do enddo(condition); // empty body
```

Chapter II.3. Atomic (critical) Section

This control structure does not move the point of execution as in selection or iteration control structures. The block of code between **atomic** and **endatomic** executes atomically in a multi-threaded program. That is, a thread will have the control of the Z47 Processor for the **entire** time while in **atomic** section. Specifically, there will be **no context switch** to another thread.

The purpose of **atomic** section is to ensure that a thread in **atomic** section will execute a block of code in its **entirety** without context switch to another thread. The use of **mutex** only blocks the threads that wish to **compete** in accessing common resources, until it is their turn to do so. However, a thread in **atomic** section will be the only thread executing.

The following example illustrates the use of **atomic** section.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

// ---------------------------------------------------------------- //
// A global integer for threads to increment.

int globalInt = 0;

// ---------------------------------------------------------------- //

enum userSignals : signalEventType {
      _SIGNAL_FirstIncrementSignal,
      _SIGNAL_SecondIncrementSignal,
      _SIGNAL_FirstTerminateSignal,
      _SIGNAL_SecondTerminateSignal,
      _SIGNAL_FirstCompletionSignal,
      _SIGNAL_SecondCompletionSignal
};

// ---------------------------------------------------------------- //
// A thread that increments globalInt.

void firstThread(int n)<thread>

      boolean done = False;

      for (;;)

// Wait until signal arrives

            if (signal ? _SIGNAL_FirstIncrementSignal)
```

```
// Entering critical section atomic..endatomic

                  atomic
                        globalInt += n;
                        if (globalInt >= 100)
                              done = True;
                        endif;
                  endatomic;

                  if (done)
                        signal <- _SIGNAL_FirstCompletionSignal;
                        break; // out of for-loop
                  endif;

            elsif (signal ? _SIGNAL_FirstTerminateSignal)
                  break; // out of for-loop
            endif;

      endfor;

end;

// ------------------------------------------------------------------ //
// A thread that increments globalInt.

void secondThread(int n)<thread>

      boolean done = False;

      for (;;)

// Wait until signal arrives

            if (signal ? _SIGNAL_SecondIncrementSignal)

// Entering critical section atomic..endatomic

                  atomic
                        globalInt += n;
                        if (globalInt >= 100)
                              done = True;
                        endif;
                  endatomic;

                  if (done)
                        signal <- _SIGNAL_SecondCompletionSignal;
                        break; // out of for-loop
                  endif;

            elsif (signal ? _SIGNAL_SecondTerminateSignal)
                  break; // out of for-loop
            endif;

      endfor;

end;
```

```
////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Starts threads with increments 1 and 2.

      firstThread(1);
      secondThread(2);

      output << "Starting value of globalInt: " << globalInt << '\n';

      do

// Tell threads to increment globalInt

            signal <- _SIGNAL_FirstIncrementSignal;
            signal <- _SIGNAL_SecondIncrementSignal;

// If one thread signals completion, terminate the other thread
// and exit the do-loop.

            if (signal ? _SIGNAL_FirstCompletionSignal)
                  signal <- _SIGNAL_SecondIncrementSignal;
                  break;
            elsif (signal ? _SIGNAL_SecondCompletionSignal)
                  signal <- _SIGNAL_FirstTerminateSignal;
                  break;
            endif;

      enddo;

      output << "Ending value of globalInt: " << globalInt << '\n';

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Starting value of globalInt: 0
Ending value of globalInt: 102
Good-bye World!
```

Chapter III. Type Definition Mechanisms

Ordinarily, we use the term definition as a means of identifying specific entities (in some universe of discourse). Those that satisfy the definition belong to a set, and all other entities belong to the complement of that set. That is, a definition is not a mechanism for constructing the entities that satisfy it.

From an engineer's perspective the introduction of a new type is a definition. However, such a definition is a set of instructions to the compiler in a predetermined format. The compiler uses the definition of a new type in order to construct objects of that type. In other words, from the compiler's perspective the definition of a new type is a mechanism for creating instances of that type. In particular, a type may include mechanisms for creating instances of other types from its own instances, and vice versa (conversions).

In this chapter we discuss the Z++ mechanisms for introducing new types. Each type definition mechanism enables engineers to have the compiler create objects that cannot be defined using other linguistic facilities in such a way that the compiler can enforce the abstractions associated with the type. **This helps understand the objects the same way that the compiler is viewing them**. It is this common view between an engineer and the compiler that reduces the possibility of errors, especially when dealing with large and complex objects.

For brevity, we also refer to a type definition mechanism as a **type constructor**. For instance, the **class** mechanism is a type constructor in that it allows an engineer to create a new type for a common way of interpreting instances of that type between the engineer and the compiler. A type defined via the **class** mechanism is also called a **class** type. This usage is to emphasize the mechanism used in defining a type, for instance that it is not a **union** or **collection** type.

Numbers and strings can describe the attributes of a large set of entities. That makes them sufficiently important to start out with these types as the fundamental types. For logical operations the **boolean** type is also important enough to be included among the fundamental types. But we need type constructors to be able to describe software entities, in a manner analogous to physics. For instance in physics, speed is described as meters per second, in standard system of units. That is, type constructors discussed in this chapter are in fact necessary.

Chapter III.1. Enumeration

Type constructor enumeration (**enum**) is essential. However, its effectiveness depends on the level of support by the language. An enumeration type allows the user to define the literal values for instances of the type, and the association of those literals with integers.

In Z++ no identifier can start with an underscore. The initial underscore is reserved for enumeration literals, which must **begin with the underscore character**.

A typical enumeration definition looks like the following.

```
enum MyNumbers {_first = 10, _second = 17, _third = 22};
```

The name of the enumeration type is `MyNumbers`.

If the numerical associations are left out, the compiler will start with 0 with successive increments of 1. At any point, if a numerical association is not specified for a literal, its numerical value will be one more than its predecessor. An instance of an enumeration is declared in the usual way.

```
MyNumbers Number = _second;
```

Section III.1.A. Privacy and Resolution of Literals

The literals defined for an enumeration type are **private** values of the type. For instance, we can define the following two enumeration types in the same context.

```
enum PositiveNumbers {_first = 10, _second = 17, _third = 22};
enum NegativeNumbers {_first = -22, _second = -17, _third = -10};
```

Notice that integer values associated with literals must be increasing. A negative number with smaller absolute value is larger.

So now we need to know how to indicate a literal from one or the other enumeration type. This is conveniently done with the resolution operator.

```
PositiveNumbers::_first
NegativeNumbers::_first
```

When you use a literal common to two enumeration types, without adorning it with its type name, the compiler will complain.

Section III.1.B. Successor and Predecessor

The increment operator `++` and decrement operator `--` are the successor and predecessor operators for objects of enumeration type, as illustrated below.

```
enum PositiveNumbers {_first = 10, _second = 17, _third = 22};
enum NegativeNumbers {_first = -22, _second = -17, _third = -10};

PositiveNumbers pNumberFirst = PositiveNumbers::_first;
PositiveNumbers pNumberSecond = ++pNumberFirst; // _second

NegativeNumbers nNumberSecond = NegativeNumbers::_second;
NegativeNumbers nNumberFirst = --nNumberSecond; // _first
```

The prefix and postfix forms of the increment/decrement operators retain their traditional meanings with regard to enumeration objects. For instance, the two instances below will have the same value `_first`.

```
PositiveNumbers pNumberFirst = PositiveNumbers::_first;
PositiveNumbers pNumberSecond = pNumberFirst++; // _first
```

The reason is that `pNumberFirst` moves to its successor after its value is used to initialize `pNumberSecond`.

Section III.1.C. Enumeration Bracket Operator

The literals of an enumeration type are precisely that: the set of literal values for instance of that type. Although enumeration literals can be **compared** with one another, an enumeration literal is not the same as its associated integer value, and unlike C, it **cannot be used in numeric computations**.

The **bracket operator (function)** applied to an enumeration literal evaluates to the numeric value associated with the literal. The **bracket operator** can also be applied to an enumeration **instance**, in which case it evaluates to the numeric value associated with the enumeration literal value of that instance. This allows using enumeration objects, just as well as enumeration literals, in numeric expression.

In the following example, both integers `firstInt` and `secondInt` are initialized to 17.

```
enum PositiveNumbers {_first = 10, _second = 17, _third = 22};
PositiveNumbers pNumberFirst = _second;

int firstInt = [pNumberFirst]; // 17
int secondInt = [_second];     // 17
```

The **bracket operator** can also be applied to the **enumeration type** itself. In this case, the bracket function evaluates to the numeric value of the **least** literal value of the type. Below `thirdInt` is initialized to 10.

```
int thirdInt = [PositiveNumbers]; // 10
```

The **double bracket operator (function)** can only be applied to an enumeration **type**, not instances of that type. The double bracket operator evaluates to the numeric value of the **largest** literal value of the type. Below `fourthInt` is initialized to 22.

```
int fourthInt = [[PositiveNumbers]]; // 22
```

The output of the following sample is the numbers 10, 17 and 22, in that order. The instance `counter` is initialized to `_first`.

```
for (PositiveNumbers counter = [PositiveNumbers];
     counter <= [[PositiveNumbers]];
     counter++)

  output << [counter] << '\n';

endfor;
```

The expression `counter <= [[PositiveNumbers]]` in the comparison section of the for-loop, compares the enumeration literal value of the object `counter` with the largest literal value of the type `PositiveNumbers`.

Section III.1.D. Extending an Enumeration Type

An important feature of enumeration type is its **extensibility**. This feature allows engineers to extend system exceptions and system signals.

The extensibility of enumeration type is **linear**. Using the notion of inheritance, linear means single-inheritance. However, since enumeration type does not admit methods, we do not use the term inheritance. Instead, we say one type is an **extension** of another one.

The following example illustrates the syntax for extending enumeration type, which is similar to **class** inheritance. Below, `firstExtendedType` extends `baseType` and the type `secondExtendedType` extends `firstExtendedType`.

```
enum baseType {_one = 1, _two = 2, _three = 3};

enum firstExtendedType : baseType {_five = 5,
                                   _eight = 8, _thirteen = 13};

enum secondExtendedType : firstExtendedType {_twentyOne = 21,
                                             _thirtyFour = 34};
```

The numeric values of an extended type must start with a number greater than the largest numeric value of the type it is extending.

An extended type includes the literals of the type it is extending. In the above example the type `secondExtendedType` includes all enumeration literals shown in the example.

When making assignments, remember that the extended type has more literals. That is, an instance of the extended type can receive a literal from its base type, but not the other way round. Here are some examples.

All four following lines are correct.

```
secondExtendedType secondInstance_1 = _one;
secondExtendedType secondInstance_2 = _eight;
secondExtendedType secondInstance_3 = _thirtyFour;
firstExtendedType firstInstance_1 = _three;
```

All three following lines will cause error.

```
firstExtendedType firstInstance_2 = _twentyOne;
baseType baseInstance_1 = _thirteen;
baseType baseInstance_2 = _thirtyFour;
```

Chapter III.2. Class

Type constructor **struct** is identical to **class** except for the default access for its members and methods. For **struct** the default is **public** and for **class** the default is **private**.

Type definition mechanism **class** supports multiple-inheritance and parameterized types (templates). It is the central mechanism for the implementation of component-oriented applications, as well as remote modules and interfacing with dynamic libraries of other languages.

Type constructor **class** supports **invariants** and **constraints**. In the following subsections we shall discuss **class** via examples in Z++. For simplicity, we may use **struct** instead of **class** in most examples.

Section III.2.A. Member Visibility

Default access for members and methods of a **class** is **private**, which can be changed to **protected** or **public**. The purpose of **protected** section will be illustrated when we discuss derivation. A **protected** or **private** member can be specified **visible** thus allowing read access without having to define a method to get the value of that member.

```
class simple
      int i<visible>;
public:
      simple(int); // a constructor
end;

simple::simple(int n)
      i = n;
end;

// using the type simple

simple s(7); // create instance s of type simple
output << s.i << '\n'; // prints 7
s.i = 10; // error: cannot access visible none-public member
```

Section III.2.B. Default Methods

Z++ compiler defines **six** default methods for a **class** type and generates code for their bodies. However, user can define each of these methods, in which case the compiler avoids its default action and uses the methods defined by user, instead.

Inside a method, the object is represented with **self**, which is a **reference** to the object owning the method. This is the equivalent of C++ **this**, which is a pointer.

Compiler default methods handle complex situations involving pointers and arrays, recursively. These will be illustrated as we proceed. These six methods are as follows.

The **default constructor** has **void** signature. This constructor sets all members of fundamental type to 0, including pointers, and calls default constructor on others.

The signature of the **copy constructor** is a constant reference to the type itself. This constructor calls the copy constructor on all members. User cannot change this. That is, a copy constructor cannot be made to call a different constructor on a member. Since the compiler default handles pointers and arrays, it is generally better to leave this constructor to the compiler.

The **default destructor** calls the default destructor on all members. **It does not delete pointers**.

The signature of the **default assignment operator** is a constant reference to the type (same as the copy constructor), and it returns a reference to the instance on which it is called. This operator handles pointers and arrays much like the copy constructor, as will be illustrated later.

The signature of the **default equality operator** is same as the default assignment operator but its return is of type **boolean**. Generally the compiler default is the desired implementation for this operator.

The signature and return of the **default inequality operator** is identical to those of the default equality operator. The compiler default implementation is the most desirable implementation for this operator, which is the negation of the default equality operator.

Section III.2.C. Default Copy Constructor

For each member, the default copy constructor will make a copy of that member, as one would do by hand. However, as you will see in the example it is only easy to say writing the code by hand, as opposed to leaving it to the compiler. The example far exceeds the possibilities of ordinary programming, just to provide the confidence in compiler's automated code generation.

For a pointer, if it is not 0, the default copy constructor will create a new object and make the pointer member of the copy point to it. Otherwise the pointer will be set to 0.

The default copy constructor is generated for you when you leave it out. In this example we are defining them for one reason. For any member **not listed after the colon** the compiler will leave the **responsibility to the user**, which probably will never be needed. However, since in our example we are listing all the members after the colon, the definition is equivalent to the default copy constructor in that you can comment out its definition and still get the same output.

Note the use of **destroy** operator in the definition of destructor. Operator **destroy** applies the **delete** operator recursively, removing all pointers and arrays and properly calling the destructors as needed.

The first example uses regular (static) arrays. We will illustrate dynamic arrays with a separate example to make it somewhat easy to understand the program.

```
// Example one, regular arrays

#include<iostream.h>
using namespace ioSpace;

//////////////////////////////////////////////////////////////////////
// Type for illustration is "struct test", defined further below.
// ---------------------------------------------------------------- //

typedef intLevelOne int*;
typedef intLevelTwo intLevelOne*;
typedef intLevelThree intLevelTwo*;

// ---------------------------------------------------------------- //
// arrayCellType is used for an array member of struct test.
// ---------------------------------------------------------------- //

struct arrayCellType
      double d;

      arrayCellType(void);
      arrayCellType(double);
      arrayCellType(const arrayCellType&);
      arrayCellType& operator=(const arrayCellType&);
end;
```

```
arrayCellType::arrayCellType(void)
      d = 33.21;
end;

arrayCellType::arrayCellType(double b)
      d = b;
end;

arrayCellType::arrayCellType(const arrayCellType& e) : d(e.d)
end;

arrayCellType& arrayCellType::operator=(const arrayCellType& e)
: d(e.d)
      return self;
end;

typedef arrayOfStruct arrayCellType[3];

// ------------------------------------------------------------------ //

typedef arrayOfStrings string[3];

// ------------------------------------------------------------------ //

typedef arrayOfPointerToDouble double***[5];
typedef pointerToArrayOfStruct arrayOfStruct*;
typedef pPointerToArrayOfStruct pointerToArrayOfStruct*;

typedef ptrToArrayOfPtrToDbl arrayOfPointerToDouble*;
typedef pPtrToArrayOfPtrToDbl ptrToArrayOfPtrToDbl*;

typedef ptrToArrayOfStgs arrayOfStrings*;
typedef pPtrToArrayOfStgs ptrToArrayOfStgs*;

typedef ptrToDbl double*;
typedef pPtrToDbl ptrToDbl*;

////////////////////////////////////////////////////////////////////
// This is the definition of the structure used for illustration.
// ------------------------------------------------------------------ //

struct test

// This is array of integer pointers
      int*** ip[3];

// This is pointer to array of structures
      arrayOfStruct*** aosPtr;

// This is pointer to array of strings
      arrayOfStrings*** aostgPtr;

// This is array of pointers to array of pointers to double
      arrayOfPointerToDouble*** ptpad[3];

// Methods
```

```
      test(void);
      ~test(void);
      test(const test&);
      test& operator=(const test&);
end;

////////////////////////////////////////////////////////////////////////
// Default constructor.
// ------------------------------------------------------------------ //

test::test(void)

// ------------------------------------------------------------------ //

      ip[0] = new intLevelTwo(new intLevelOne(new int(3)));
      ip[1] = new intLevelTwo(new intLevelOne(new int(5)));
      ip[2] = new intLevelTwo(new intLevelOne(new int(7)));

// ------------------------------------------------------------------ //
// The following is really one single line.
// ------------------------------------------------------------------ //

      aosPtr = new
            pPointerToArrayOfStruct(new pointerToArrayOfStruct(
            new arrayOfStruct(97.47)));

// ------------------------------------------------------------------ //

      arrayOfStrings* aos = new arrayOfStrings;
      (*aos)[0] = "First Cell";
      (*aos)[1] = "Second Cell";
      (*aos)[2] = "Third Cell";

      aostgPtr = new pPtrToArrayOfStgs(new ptrToArrayOfStgs(aos));

// ------------------------------------------------------------------ //
// Each of the following are really single lines.
// ------------------------------------------------------------------ //

      ptpad[0] = new pPtrToArrayOfPtrToDbl(
            new ptrToArrayOfPtrToDbl(new arrayOfPointerToDouble));

      (***ptpad[0])[0] = new pPtrToDbl(
            new ptrToDbl(new double(11.13)));

      (***ptpad[0])[1] = new pPtrToDbl(
            new ptrToDbl(new double(22.23)));

      (***ptpad[0])[2] = new pPtrToDbl(
            new ptrToDbl(new double(33.33)));

      (***ptpad[0])[3] = new pPtrToDbl(
            new ptrToDbl(new double(31.31)));

      (***ptpad[0])[4] = new pPtrToDbl(
            new ptrToDbl(new double(32.32)));
```

```
      ptpad[1] = new pPtrToArrayOfPtrToDbl(
            new ptrToArrayOfPtrToDbl(new arrayOfPointerToDouble));

      (***ptpad[1])[0] = new pPtrToDbl(
            new ptrToDbl(new double(44.43)));

      (***ptpad[1])[1] = new pPtrToDbl(
            new ptrToDbl(new double(55.53)));

      (***ptpad[1])[2] = new pPtrToDbl(
            new ptrToDbl(new double(66.63)));

      (***ptpad[1])[3] = new pPtrToDbl(
            new ptrToDbl(new double(61.61)));

      (***ptpad[1])[4] = new pPtrToDbl(
            new ptrToDbl(new double(62.62)));

      ptpad[2] = new pPtrToArrayOfPtrToDbl(
            new ptrToArrayOfPtrToDbl(new arrayOfPointerToDouble));

      (***ptpad[2])[0] = new pPtrToDbl(
            new ptrToDbl(new double(77.73)));

      (***ptpad[2])[1] = new pPtrToDbl(
            new ptrToDbl(new double(88.83)));

      (***ptpad[2])[2] = new pPtrToDbl(
            new ptrToDbl(new double(99.93)));

      (***ptpad[2])[3] = new pPtrToDbl(
            new ptrToDbl(new double(91.91)));

      (***ptpad[2])[4] = new pPtrToDbl(
            new ptrToDbl(new double(92.92)));

end;

////////////////////////////////////////////////////////////////////////
// Destructor. Operator destroy will recursively delete, pointers
// arrays.
// ------------------------------------------------------------------ //

test::~test(void)
      output << "\nIn test::~test(void)....\n";
      destroy ip;
      destroy aosPtr;
      destroy aostgPtr;
      destroy ptpad;
end;

////////////////////////////////////////////////////////////////////////
// Copy constructor. Members not listed after colon will not be
// processed by compiler, and user will be responsible for them.
// ------------------------------------------------------------------ //
// The definition here is equivalent to default copy constructor, in
```

```
// that we are listing all members for compiler to process for us.
// You can comment out the definition (and its prototype), and get
// exactly the same output.
// ------------------------------------------------------------------ //

test::test(const test& e)
: ip(e.ip), aosPtr(e.aosPtr), aostgPtr(e.aostgPtr), ptpad(e.ptpad)
end;

//////////////////////////////////////////////////////////////////////
// Assignment. Members not listed after colon will not be processed
// by compiler, and user will be responsible for them.
// ------------------------------------------------------------------ //
// The definition here is equivalent to default assignment operator
// in that we are listing all members for compiler to process for us.
// You can comment out the definition (and its prototype), and get
// exactly the same output.
// ------------------------------------------------------------------ //

test& test::operator=(const test& e)
: ip(e.ip), aosPtr(e.aosPtr), aostgPtr(e.aostgPtr), ptpad(e.ptpad)
      return self;
end;

//////////////////////////////////////////////////////////////////////
// A global function for testing copy constructor.
// ------------------------------------------------------------------ //

void fun(test s)

      output << "\nIn fun, printing values of argument s.\n";

      for (int i = 0; i < 3; i++)

            output << "In fun, ***s.ip[i] is: " << ***s.ip[i] << '\n';

            output << "In fun, (***s.aosPtr)[i].d is: " <<
                  (***s.aosPtr)[i].d << '\n';

            output << "In fun, (***s.aostgPtr)[i] is: " <<
                  (***s.aostgPtr)[i] << '\n';

            for (int j = 0; j < 5; j++)
                  output << "In fun, ***(***s.ptpad[i])[j] is: " <<
                        ***(***s.ptpad[i])[j] << '\n';
            endfor;

      endfor;

      output << "\nLeaving fun().\n";
end;

//////////////////////////////////////////////////////////////////////
//                    The Main Entry Point
// ------------------------------------------------------------------ //
//////////////////////////////////////////////////////////////////////
```

```
entry void main(void)
      output << "Hello World!\n";

      output << "In main, creating object t.\n";

      test t;

      output << "In main, passing object t to fun().\n";

      fun(t);

      output << "In main, creating object g.\n";

      test g;

      output << "In main, changing values of object t.\n";

      for (int i = 0; i < 3; i++)
            ***t.ip[i] = i;
            (***t.aosPtr)[i].d = double(i) * 3.7;
            (***t.aostgPtr)[i] = "Changed in loop";

            for (int j = 0; j < 5; j++)
                  ***(***t.ptpad[i])[j] *= (i*j);
            endfor;

      endfor;

      output << "In main, making assignment g = t.\n";

      g = t;

      output << "\nIn main, printing values of g after assignment.\n";

      for (int i = 0; i < 3; i++)

            output << "In main, ***g.ip[i] is: " << ***g.ip[i] << '\n';

            output << "In main, (***g.aosPtr)[i].d is: " <<
                  (***g.aosPtr)[i].d << '\n';

            output << "In main, (***g.aostgPtr)[i] is: " <<
                  (***g.aostgPtr)[i] << '\n';

            for (int j = 0; j < 5; j++)
                  output << "In main, ***(***g.ptpad[i])[j] is: " <<
                        ***(***g.ptpad[i])[j] << '\n';

            endfor;

      endfor;

      output << "Good-bye World!\n";
end;
```

The output of this program is as follows.

```
Hello World!
In main, creating object t.
In main, passing object t to fun().

In fun, printing values of argument s.
In fun, ***s.ip[i] is: 3
In fun, (***s.aosPtr)[i].d is: 97.47
In fun, (***s.aostgPtr)[i] is: First Cell
In fun, ***(***s.ptpad[i])[j] is: 11.13
In fun, ***(***s.ptpad[i])[j] is: 22.23
In fun, ***(***s.ptpad[i])[j] is: 33.33
In fun, ***(***s.ptpad[i])[j] is: 31.31
In fun, ***(***s.ptpad[i])[j] is: 32.32
In fun, ***s.ip[i] is: 5
In fun, (***s.aosPtr)[i].d is: 97.47
In fun, (***s.aostgPtr)[i] is: Second Cell
In fun, ***(***s.ptpad[i])[j] is: 44.43
In fun, ***(***s.ptpad[i])[j] is: 55.53
In fun, ***(***s.ptpad[i])[j] is: 66.63
In fun, ***(***s.ptpad[i])[j] is: 61.61
In fun, ***(***s.ptpad[i])[j] is: 62.62
In fun, ***s.ip[i] is: 7
In fun, (***s.aosPtr)[i].d is: 97.47
In fun, (***s.aostgPtr)[i] is: Third Cell
In fun, ***(***s.ptpad[i])[j] is: 77.73
In fun, ***(***s.ptpad[i])[j] is: 88.83
In fun, ***(***s.ptpad[i])[j] is: 99.93
In fun, ***(***s.ptpad[i])[j] is: 91.91
In fun, ***(***s.ptpad[i])[j] is: 92.92

Leaving fun().
In main, creating object g.
In main, changing values of object t.
In main, making assignment g = t.

In main, printing values of g after assignment.
In main, ***g.ip[i] is: 0
In main, (***g.aosPtr)[i].d is: 0
In main, (***g.aostgPtr)[i] is: Changed in loop
In main, ***(***g.ptpad[i])[j] is: 0
In main, ***(***g.ptpad[i])[j] is: 0
In main, ***(***g.ptpad[i])[j] is: 0
In main, ***(***g.ptpad[i])[j] is: 0
In main, ***(***g.ptpad[i])[j] is: 0
In main, ***g.ip[i] is: 1
In main, (***g.aosPtr)[i].d is: 3.7
In main, (***g.aostgPtr)[i] is: Changed in loop
In main, ***(***g.ptpad[i])[j] is: 0
In main, ***(***g.ptpad[i])[j] is: 55.53
In main, ***(***g.ptpad[i])[j] is: 133.26
In main, ***(***g.ptpad[i])[j] is: 184.83
In main, ***(***g.ptpad[i])[j] is: 250.48
In main, ***g.ip[i] is: 2
In main, (***g.aosPtr)[i].d is: 7.4
In main, (***g.aostgPtr)[i] is: Changed in loop
```

```
In main, ***(***g.ptpad[i])[j] is: 0
In main, ***(***g.ptpad[i])[j] is: 177.66
In main, ***(***g.ptpad[i])[j] is: 399.72
In main, ***(***g.ptpad[i])[j] is: 551.46
In main, ***(***g.ptpad[i])[j] is: 743.36
Good-bye World!

In test::~test(void)....

In test::~test(void)....
```

Section III.2.D. Default Assignment Operator

The example of previous subsection included the default assignment operator for cases when types of members involve sequences of regular arrays and pointers. The example in this subsection will also include the copy constructor, and the assignment operator. However, in this example we introduce dynamic arrays.

For all constructors, except the copy constructor, the compiler will require that you initialize a dynamic array in the list following the colon. This is because the compiler needs to create the dynamic array and for that it needs information about the sizes of dimensions of the array. This only applies when the type of member is a dynamic array, not a pointer to a dynamic array.

In presence of dynamic arrays as members, the compiler cannot generate code for the default assignment operator, and will report an error if you do not define one. That is because the compiler will have to change the object being assigned to in unusual ways. For instance, when the sizes of dynamic arrays are not the same, the object being assigned to will have to change structurally, and without any notification. Therefore, the implementation of default assignment operator is left to the user. In most cases you probably will not be assigning such objects to one another. In that case your implementation can be reduced to returning the **self**, and perhaps raising an exception if you think the use of assignment would be an error.

Fortunately, the default comparison operators work for dynamic arrays, as well. Keep in mind that instances of classes with pointer members may not be equal even after assigning one to the other. The reason is that their pointer members may be pointing to different objects, which in fact are equal. However, comparison operators compare the members, not what a pointer member is pointing to.

In general, even though the default copy constructor works correctly, unless really necessary you would pass such large objects by reference.

The use of dynamic arrays as members of a class is quite simple, though. The example uses complex sequence of pointers and combinations of arrays for types of members in

order to demonstrate the ability of the compiler in generating code for unusually complex situations. The example is not meant to be comprehendible.

Important Remark. For a special case regarding assignment of union members of a **class** see Union Assignment.

```
// Example two, dynamic arrays

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////
// The copy constructor is illustrated by calling the global function
// testFunValue(), on a struct with dynamic arrays/pointers.
// The illustrating type is struct DAmember.
//------------------------------------------------------------------ //

//------------------------------------------------------------------ //
// Values for CollectionArgument
//------------------------------------------------------------------ //

enum argumentEnum {_one, _two, _three};

//------------------------------------------------------------------ //

struct ArgumentOne
      int i;

      ArgumentOne(void);
end;

ArgumentOne::ArgumentOne(void)
      i = 97;
end;

//------------------------------------------------------------------ //

struct ArgumentTwo
      double d;

      ArgumentTwo(void);
end;

ArgumentTwo::ArgumentTwo(void)
      d = 97.47;
end;

//------------------------------------------------------------------ //

struct ArgumentThree
      short s;

      ArgumentThree(void);
end;
```

```
ArgumentThree::ArgumentThree(void)
      s = 7;
end;

//------------------------------------------------------------------- //
// Definition of CollectionArgument
//------------------------------------------------------------------- //

collection CollectionArgument<argumentEnum> {
      _one<ArgumentOne>,
      _two<ArgumentTwo>,
      _three<ArgumentThree>
};

//------------------------------------------------------------------- //
// A struct to be used as member.
//------------------------------------------------------------------- //

struct forMember
      double d;

      forMember(void);
      forMember(double);
end;

forMember::forMember(void)
      d = 97.47;
end;

forMember::forMember(double b)
      d = b;
end;

//------------------------------------------------------------------- //

int m = 5;

typedef dynPointerArrayStruct forMember*[];

// dapCollection is dyn array of pointers to dyn array of collections.
typedef dynArrayOfCollection CollectionArgument[];
typedef dapCollection dynArrayOfCollection*[];

// Static array [3] of pointer/pointer to dyn array of pointer/pointer
// to struct
typedef dynPointerPointerStruct forMember**[];
typedef staticPP dynPointerPointerStruct*[3];

typedef fmPtrType forMember*;

//------------------------------------------------------------------- //
//            Defining main type: struct DAmember
//------------------------------------------------------------------- //
// dpas is a dynamic array of pointers to structures.
//------------------------------------------------------------------- //
// dap is a dynamic array of pointers to dynamic array of collections,
// illustrating recursion on dynamic-array/pointer/dynamic-array.
```

```
//------------------------------------------------------------------- //
// dac is a pointer member to a dynamic array of collection.
// Here, member starts out as a pointer to dynamic arrays.
//------------------------------------------------------------------- //
// spp is static array of pointers to dynamic array of pointer/pointer
// to collections.
//------------------------------------------------------------------- //

struct DAmember

      dynPointerArrayStruct dpas;
      dapCollection dap;
      dynArrayOfCollection* dac;
      staticPP spp;

// Methods

      DAmember(void);
      DAmember(int r, int s, int t);
      //DAmember(const DAmember&); // using default copy constructor
      DAmember& operator=(const DAmember&); // implemented
end;

//------------------------------------------------------------------- //
// Default constructor.
//------------------------------------------------------------------- //

DAmember::DAmember(void) : dpas[m](), dap[3]()

      for (int counter = 0; counter < m; counter++)
            dpas[counter] = new forMember(11.33);
      endfor;

      dap[0] = new dynArrayOfCollection[2]();
      dap[1] = new dynArrayOfCollection[3]();
      dap[2] = new dynArrayOfCollection[4]();

      dac = new dynArrayOfCollection[4]();

      spp[0] = new dynPointerPointerStruct[2]();
      (*spp[0])[0] = new fmPtrType(new forMember(1.7));
      (*spp[0])[1] = new fmPtrType(new forMember(1.9));

      spp[1] = new dynPointerPointerStruct[3]();
      (*spp[1])[0] = new fmPtrType(new forMember(2.3));
      (*spp[1])[1] = new fmPtrType(new forMember(2.7));
      (*spp[1])[2] = new fmPtrType(new forMember(2.9));

      spp[2] = new dynPointerPointerStruct[4]();
      (*spp[2])[0] = new fmPtrType(new forMember(3.3));
      (*spp[2])[1] = new fmPtrType(new forMember(3.5));
      (*spp[2])[2] = new fmPtrType(new forMember(3.7));
      (*spp[2])[3] = new fmPtrType(new forMember(3.9));
end;

//------------------------------------------------------------------- //
// Constructor. r, s, t come in as 2,3,4, respectively.
```

```
//------------------------------------------------------------------- //

DAmember::DAmember(int r, int s, int t) : dpas[m](), dap[3]()

      for (int counter = 0; counter < m; counter++)
            dpas[counter] = new forMember(55.99);
      endfor;

      dap[0] = new dynArrayOfCollection[r]();
      dap[1] = new dynArrayOfCollection[s]();
      dap[2] = new dynArrayOfCollection[t]();

      dac = new dynArrayOfCollection[t]();

      spp[0] = new dynPointerPointerStruct[r]();
      for (int counter = 0; counter < r; counter++)
            (*spp[0])[counter] = new fmPtrType(new forMember(77.11));
      endfor;

      spp[1] = new dynPointerPointerStruct[s]();
      for (int counter = 0; counter < s; counter++)
            (*spp[1])[counter] = new fmPtrType(new forMember(22.55));
      endfor;

      spp[2] = new dynPointerPointerStruct[t]();
      for (int counter = 0; counter < t; counter++)
            (*spp[2])[counter] = new fmPtrType(new forMember(44.22));
      endfor;

end;

//------------------------------------------------------------------- //
// Illustrating the implementation of operator assign in presence
// of dynamic array members.
//------------------------------------------------------------------- //
// To keep the code understandable only when sizes of dimensions
// are same the assignment is done. However, when dimensions are
// not same it is done similarly. Destroy the array, and recreate
// it like its argument as is done when sizes are the same.
//------------------------------------------------------------------- //

DAmember& DAmember::operator=(const DAmember& e)

  int first, second, firstDimSize, secondDimSize;

// dpas is a dynamic array of pointers to structures.

  firstDimSize = size(dpas, 0);

  if (firstDimSize == size(e.dpas, 0))
    for (first = 0; first < firstDimSize; first++)

      if (dpas[first]) destroy dpas[first]; endif;

      if (e.dpas[first])
        dpas[first] = new forMember(*e.dpas[first]);
        else dpas[first] = 0;
```

```
      endif;

    endfor;
  endif;

// dap is a dynamic array of pointers to dynamic array of collections.

  firstDimSize = size(dap, 0);

    if (firstDimSize == size(e.dap, 0))
      for (first = 0; first < firstDimSize; first++)

        if (dap[first])

          if (e.dap[first])
            secondDimSize = size(*dap[first], 0);
            if (secondDimSize == size(*e.dap[first], 0))
              destroy dap[first];
              for (second = 0; second < secondDimSize; second++)
                dap[first] = new
                   dynArrayOfCollection[secondDimSize]
                   ((*e.dap[first])[second]);
              endfor;
             endif;
            else
              destroy dap[first];
              dap[first] = 0;
            endif;

        elsif (e.dap[first])

          secondDimSize = size(*e.dap[first], 0);
          for (second = 0; second < secondDimSize; second++)
           dap[first] = new
            dynArrayOfCollection[secondDimSize]((*e.dap[first])[second]);
          endfor;

        endif;

    endfor;
  endif;

// dac is a pointer member to a dynamic array of collection.

      if (dac)

            if (e.dac)

                  firstDimSize = size(*dac, 0);
                  if (firstDimSize == size(*e.dac, 0))

                        destroy dac;
                        dac = new dynArrayOfCollection[firstDimSize]();
                        for (first = 0; first < firstDimSize; first++)
                              (*dac)[first] = (*e.dac)[first];
                        endfor;
                  endif;
```

```
            else
                  destroy dac;
                  dac = 0;
            endif;

      elsif (e.dac)
            firstDimSize = size(*e.dac, 0);
            dac = new dynArrayOfCollection[firstDimSize]();
            for (first = 0; first < firstDimSize; first++)
                  (*dac)[first] = (*e.dac)[first];
            endfor;
      endif;

// spp is static array of pointers to dynamic array of pointer/pointer
// to collections.

  firstDimSize = 3; // size of static array is known

    for (first = 0; first < firstDimSize; first++)

      if (spp[first])

          if (e.spp[first])

            secondDimSize = size(*spp[first], 0);
            if (secondDimSize == size(*e.spp[first], 0))
                  destroy spp[first];
                  spp[first] = new
                    dynPointerPointerStruct[secondDimSize]
                    (new fmPtrType(new forMember()));

                  for (second = 0; second < secondDimSize; second++)
                    (**(*spp[first])[second]) =
                      (**(*e.spp[first])[second]);
                  endfor;

            endif;

          else
            destroy spp[first];
            spp[first] = 0;
          endif;

        elsif (e.spp[first])

          secondDimSize = size(*spp[first], 0);
          spp[first] = new dynPointerPointerStruct[secondDimSize](
            new fmPtrType(new forMember()));

          for (second = 0; second < secondDimSize; second++)
            (**(*spp[first])[second]) = (**(*e.spp[first])[second]);
          endfor;

        endif;

  endfor;
```

```
  return self;
end;

////////////////////////////////////////////////////////////////////////
// Global function for illustrating default copy constructor.
//-------------------------------------------------------------------- //

void testFunValue(DAmember dam)

      output << "In testFunValue(), testing copy ctor...\n\n";

      output << "Pointer to Structure...\n\n";

      int dimOne = size(dam.dpas, 0);
      for (int first = 0; first < dimOne; first++)
            output << "(*dam.dpas[first]).d is: " <<
                  (*dam.dpas[first]).d << '\n';
      endfor;

      output << "\nDyn Array of pointer to Dyn array of
collection...\n\n";

      output << "(*dam.dap[0])[1][_one].i is: " <<
            (*dam.dap[0])[1][_one].i << '\n';
      output << "(*dam.dap[1])[0][_two].d is: " <<
            (*dam.dap[1])[0][_two].d << '\n';
      output << "(*dam.dap[2])[3][_three].s is: " <<
            (*dam.dap[2])[3][_three].s << '\n';

      output << "\nPointer to Dyn array of collection...\n\n";

      output << "(*dam.dac)[0][_one].i is: " <<
            (*dam.dac)[0][_one].i << '\n';
      output << "(*dam.dac)[1][_two].d is: " <<
            (*dam.dac)[1][_two].d << '\n';
      output << "(*dam.dac)[2][_three].s is: " <<
            (*dam.dac)[2][_three].s << '\n';
      output << "(*dam.dac)[3][_three].s is: " <<
            (*dam.dac)[3][_three].s << '\n';

      output << "\nThe ridiculous case...\n\n";

      output << "(**(*dam.spp[0])[0]).d is: " <<
            (**(*dam.spp[0])[0]).d << '\n';
      output << "(**(*dam.spp[1])[1]).d is: " <<
            (**(*dam.spp[1])[1]).d << '\n';
      output << "(**(*dam.spp[2])[2]).d is: " <<
            (**(*dam.spp[2])[2]).d << '\n';

end;

////////////////////////////////////////////////////////////////////////
//                  Main Entry Point
//-------------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////
```

```
entry void main(void)

      output << "Hello World!\n";

//---------------------------------------------------------------------- //

      DAmember DAM;

      output << "\nCalling testFunValue(DAM)\n\n";

      testFunValue(DAM);

//---------------------------------------------------------------------- //

      DAmember SAM(2, 3, 4);

      output << "\nCalling testFunValue(SAM)\n\n";

      testFunValue(SAM);

//---------------------------------------------------------------------- //

      DAM = SAM;

      output << "\nCalling testFunValue(DAM) after assignment DAM =
SAM.\n\n";

      testFunValue(DAM);

//---------------------------------------------------------------------- //

      output << "Good-bye World!\n";
end;
```

The output of this program is as follows.

```
Hello World!

Calling testFunValue(DAM)

In testFunValue(), testing copy ctor...

Pointer to Structure...

(*dam.dpas[first]).d is: 11.33
(*dam.dpas[first]).d is: 11.33
(*dam.dpas[first]).d is: 11.33
(*dam.dpas[first]).d is: 11.33
(*dam.dpas[first]).d is: 11.33

Dyn Array of pointer to Dyn array of collection...

(*dam.dap[0])[1][_one].i is: 97
(*dam.dap[1])[0][_two].d is: 97.47
(*dam.dap[2])[3][_three].s is: 7
```

```
Pointer to Dyn array of collection...

(*dam.dac)[0][_one].i is: 97
(*dam.dac)[1][_two].d is: 97.47
(*dam.dac)[2][_three].s is: 7
(*dam.dac)[3][_three].s is: 7

The ridiculous case...

(**(*dam.spp[0])[0]).d is: 1.7
(**(*dam.spp[1])[1]).d is: 2.7
(**(*dam.spp[2])[2]).d is: 3.7

Calling testFunValue(SAM)

In testFunValue(), testing copy ctor...

Pointer to Structure...

(*dam.dpas[first]).d is: 55.99
(*dam.dpas[first]).d is: 55.99
(*dam.dpas[first]).d is: 55.99
(*dam.dpas[first]).d is: 55.99
(*dam.dpas[first]).d is: 55.99

Dyn Array of pointer to Dyn array of collection...

(*dam.dap[0])[1][_one].i is: 97
(*dam.dap[1])[0][_two].d is: 97.47
(*dam.dap[2])[3][_three].s is: 7

Pointer to Dyn array of collection...

(*dam.dac)[0][_one].i is: 97
(*dam.dac)[1][_two].d is: 97.47
(*dam.dac)[2][_three].s is: 7
(*dam.dac)[3][_three].s is: 7

The ridiculous case...

(**(*dam.spp[0])[0]).d is: 77.11
(**(*dam.spp[1])[1]).d is: 22.55
(**(*dam.spp[2])[2]).d is: 44.22

Calling testFunValue(DAM) after assignment DAM = SAM.

In testFunValue(), testing copy ctor...

Pointer to Structure...

(*dam.dpas[first]).d is: 55.99
(*dam.dpas[first]).d is: 55.99
(*dam.dpas[first]).d is: 55.99
(*dam.dpas[first]).d is: 55.99
(*dam.dpas[first]).d is: 55.99

Dyn Array of pointer to Dyn array of collection...
```

```
(*dam.dap[0])[1][_one].i is: 97
(*dam.dap[1])[0][_two].d is: 97.47
(*dam.dap[2])[3][_three].s is: 7

Pointer to Dyn array of collection...

(*dam.dac)[0][_one].i is: 97
(*dam.dac)[1][_two].d is: 97.47
(*dam.dac)[2][_three].s is: 7
(*dam.dac)[3][_three].s is: 7

The ridiculous case...

(**(*dam.spp[0])[0]).d is: 44.22
(**(*dam.spp[1])[1]).d is: 44.22
(**(*dam.spp[2])[2]).d is: 44.22
Good-bye World!
```

Section III.2.E. Calls on Arrays of Objects

Methods, including (overloaded) operators of a **class** can be invoked on arrays of that **class**. The kind of arrays can be regular or dynamic. It is important to note that an array call always returns a **reference** to the array object on which the **call is made**. All objects returned from individual calls to cells of the array are destroyed. This makes it possible to make a sequence of calls on an array.

In the following example we are also passing arrays as arguments. All passes are by value for demonstration purposes. Passing arrays by reference in this example would be more efficient.

The output of this example is about 400 lines, showing the destruction of objects created in pass by value, and the objects returned from calls and declared objects.

```
// Example

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////
// Member will be used as a member of struct Top for demonstration
// of nesting in array calls.
//-----------------------------------------------------------------//

struct Member
      int* ip;

      Member(void);
      ~Member(void);
end;

Member::Member(void)
      ip = new int(7);
```

```
end;

Member::~Member(void)
      output << "\nIn Member::~Member(void) deleting ip: " <<
         *ip << '\n';

      delete ip;
end;

//-----------------------------------------------------------------------//
// Typedefs for parameter type specification.
//-----------------------------------------------------------------------//

typedef arrayMember Member[3];
typedef dArrayMember Member[];

//-----------------------------------------------------------------------//
// The structure on which array calls will be made.
//-----------------------------------------------------------------------//

struct Top
      Member m;

      ~Top(void);

      Member operator+(void);

      Member operator+=(arrayMember);
      Member operator+=(dArrayMember);

      Member operator*=(arrayMember);
      Member operator*=(dArrayMember);

      Member methodWhole(arrayMember);
      Member methodWhole(dArrayMember);

      Member methodTotal(arrayMember);
      Member methodTotal(dArrayMember);
end;

Top::~Top(void)
      output << "\nIn Top::~Top(void).\n";
end;

Member Top::operator+(void)
//    *m.ip++; // this will increment the pointer
      *m.ip += 1; // increments object pointed to
      return m;
end;

Member Top::operator+=(arrayMember am)
      *m.ip += *am[0].ip;
      return m;
end;

Member Top::operator+=(dArrayMember dm)
      *m.ip += *dm[0].ip;
```

```
	return m;
end;

Member Top::operator*=(arrayMember am)
	*m.ip *= 2;
	return m;
end;

Member Top::operator*=(dArrayMember dm)
	*m.ip *= 2;
	return m;
end;

//-----------------------------------------------------------------//

Member Top::methodWhole(arrayMember am)
	*m.ip *= 3;
	return m;
end;

Member Top::methodWhole(dArrayMember dm)
	*m.ip *= 3;
	return m;
end;

Member Top::methodTotal(arrayMember am)
	*m.ip *= 5;
	return m;
end;

Member Top::methodTotal(dArrayMember dm)
	*m.ip *= 5;
	return m;
end;

////////////////////////////////////////////////////////////////////
//----------------------------------------------------------------//
////////////////////////////////////////////////////////////////////

entry void main(void) // main entry point

	output << "Hello World!\n";

//-----------------------------------------------------------------//

	int dim = 3;

//-----------------------------------------------------------------//
// amt is the array on which all calls are made. Changing dim to
// a number, say 3, will make it a regular array.
//-----------------------------------------------------------------//

	Top amt[dim];

	Member amm[3]; // regular array argument
	Member dmm[dim]; // dynamic array argument
```

```
//-------------------------------------------------------------------//
// Nested sequence of calls including operators and methods.
// Arguments passed to calls are regular/dynamic arrays.
//-------------------------------------------------------------------//

      (amt.methodWhole(amm)) *= dmm;
      (amt.methodWhole(dmm)) += amm;

      +((amt *= dmm) += dmm);

      (amt *= amm) += amm;

      (amt.methodTotal(dmm)).methodWhole(amm);

      (amt.methodTotal(dmm) += amm).methodWhole(amm) *= dmm;

//-------------------------------------------------------------------//

      output << "Good-bye World!\n";
end;
```

Section III.2.F. Array Constructions

Arrays of objects of class-type can be created using any constructor for the **class**, as illustrated in the following example.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////

class cellType
      long l <visible>;
      string s <visible>;
      double d <visible>;

public:

      cellType(long, string, double);
end;

cellType::cellType(long ll, string ss, double dd)
      l = ll;
      s = ss;
      d = dd;
end;

// ------------------------------------------------------------------ //

typedef stArray cellType[3];

////////////////////////////////////////////////////////////////////////
```

```
// ------------------------------------------------------------------ //
//////////////////////////////////////////////////////////////////////

entry void main(void)

  output << "Hello World!\n";

// ------------------------------------------------------------------ //
// A two-dimensional dynamic array: dynArray.

  int dimOne = 3, dimTwo = 4;

  cellType dynArray[dimOne][dimTwo](47, "I am string.", 47.97);

  output << "Member of Dynamic array of cellType...\n";

  for (int i = 0; i < size(dynArray, 0); i++)
    for (int j = 0; j < size(dynArray, 1); j++)

      output << "\ni and j are: " << i << "," << j << '\n';
      output << "Member l is: " << dynArray[i][j].l << '\n';
      output << "Member s is: " << dynArray[i][j].s << '\n';
      output << "Member d is: " << dynArray[i][j].d << '\n';

    endfor;
  endfor;

// ------------------------------------------------------------------ //
// A regular array via the new operator: aptr.

  stArray* aptr = new stArray(77, "Regular array member.", 77.77);

  output << "\nMember of Regular array of cellType...\n";

  for (int i = 0; i < 3; i++)
    output << "\nMember l is: " << (*aptr)[i].l << '\n';
    output << "Member s is: " << (*aptr)[i].s << '\n';
    output << "Member d is: " << (*aptr)[i].d << '\n';
  endfor;

// ------------------------------------------------------------------ //

  output << "\nGood-bye World!\n";

end;
```

Section III.2.G. Dynamic Array Members

The **typedef** for a dynamic array does not accept the size of dimensions. The result is a degenerate array that can become a dynamic array when its sizes are provided in a declaration. This conveniently allows for a dynamic array member to receive its sizes through the constructor. The following example illustrates this.

The example uses nested structures and arrays to show the ease of constructing structures with dynamic array members. In this example the assignment operator is not used, but is required as discussed in Default Assignment Operator. We have defined an essentially blank default assignment operator.

The output of the example is 9.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////

struct BaseLeft
      int i;

      BaseLeft(void);
      BaseLeft(int);
end;

BaseLeft::BaseLeft(void)
      i = 0;
end;

BaseLeft::BaseLeft(int n)
      i = n;
end;

// ------------------------------------------------------------------ //

struct Middle
      BaseLeft b[3];

      Middle(void);
      Middle(int);
      void operator+(int k = 13);
end;

Middle::Middle(void)
      for (int j = 0; j < 3; j++) b[j].i = 7*j; endfor;
end;

Middle::Middle(int k)
      for (int j = 0; j < 3; j++) b[j].i = 7*j + k; endfor;
```

```
end;

void Middle::operator+(int k)
      for (int j = 0; j < 3; j++) b[j].i = k; endfor;
end;

// ---------------------------------------------------------------- //
// Used for dimensions of dynamic array for the
// default constructor of DynamicArrayMember.

int m = 5, n = 7;

// ---------------------------------------------------------------- //
// For type of member dam of DynamicArrayMember

typedef DynamicArrayOfMiddle Middle[][];

// ---------------------------------------------------------------- //
// A dynamic array member must be declared via typedef as a
// degenerate kind.
// Then, the dimensions are provided via constructors, as shown below.

struct DynamicArrayMember
      DynamicArrayOfMiddle dam;

      DynamicArrayMember(void);
      DynamicArrayMember(int, int, int);
      DynamicArrayMember& operator=(const DynamicArrayMember&);

end;

// A dynamic array member must be initialized in a manner similar to
// const members. The sizes of dimensions must be provided, even
// in case of default constructor.
// However, the literal 47 is a call to constructor for cells of
// the array. If not put there, then the default constructor
// will initialize the cells.
// Remember that you still need parentheses, like this dam[5][7](),
// to invoke the default constructor for initializing the array cells.

DynamicArrayMember::DynamicArrayMember(void) : dam[5][7](47)
end;

DynamicArrayMember::DynamicArrayMember(int m, int n, int k)
    : dam[m][n](k)
end;

DynamicArrayMember&
DynamicArrayMember::operator=(const DynamicArrayMember& e)
      return self;
end;

////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //

entry void main(void)
```

```
// The following would invoke the default constructor,
// equivalent to: D(5, 7, 47)
//    DynamicArrayMember D;

      DynamicArrayMember D(2, 3, 7);

// (D.dam + 9) will change all cells to 9 before the cell is
// reached for output, and prints 9 for the value of member i.

      output << (D.dam + 9)[0][0].b[0].i << '\n';

/* This will print a long list of 9 indicating that the call
** "D.dam + 9" actually changed all cells.

      for (int i = 0; i < size(D.dam, 0); i++)
            for (int j = 0; j < size(D.dam, 1); j++)
                  for (int k = 0; k < 3; k++)
                        output << D.dam[i][j].b[k].i << '\n';
                  endfor;
            endfor;
      endfor;
*/

end;
```

Section III.2.H. Inheritance

The most important use of derivation of a **class** from other classes is in its use in polymorphism. The type constructor collection facilitates the use of methods common to a **set of classes** without being related via inheritance. However, here too polymorphism enhances the utility of **collection** type constructor.

Examples of inheritance are sprinkled all over this chapter. Here, we only discuss a few simple rules associated with the mechanism of inheritance for defining a new **class** derived from previously defined **class** types.

Z++ derivation is **multiple-inheritance**. The default for derivation is **public**, and it may be specified **private**, as shown below.

```
class newType : firstType, private secondType
// Body of class
end;
```

In the above example, firstType and secondType are assumed to have been defined before the definition of newType. The derivation from firstType is **public**.

A privately derived type, secondType in the above example, becomes **private** part of newType. Than means, users of newType will have no access to **public** items of secondType. Furthermore, any type deriving (inheriting) from newType will have **no** access to any item of secondType.

However, since firstType is publicly derived, **public** methods (and members) of firstType remain **public** relative to newType. Any user of newType has access to **public** items of firstType.

In a derivation, the **protected** methods/members play a special role. Protected items are **private** relative to users of a **class**. However, in a derivation they are accessible to the derived **class**. In the above example, **protected** items of secondType remain **protected** items of newType, and are therefore accessible to methods of newType.

For operators reaching bases and their members see Scope Operator and Structure Operators.

Section III.2.I. Class Invariants

The purpose of invariants is to ensure that an object remains in an intended desirable state. The initial state of an object is set by, the **constructor** used in its creation. Thereafter, the state of the object can change by invoking the **public methods** of its **class**.

The state of an instance of a **class** is the state of its members, and members of its bases, recursively. Invariants are **boolean** expressions involving members of the **class** in which they are specified. Furthermore, invariants are **inherited**.

Invariants are verified **at completion of execution of public methods**. One of two things can be specified as a course of action for the violation of an invariant. One can **raise** an exception, or invoke a **private** method of the **class** with the intention of rectifying the state of the object. We refer to the **private** method invoked as a response to violation of an invariant as **trigger**. That is, the violation **triggers** the invocation of the specified method.

After completion of its execution, a trigger returns to its caller unless its execution raises an exception.

The location for specifying invariants within a **class** definition is not significant in any way. For readability, invariants should come around the start of a **class** definition.

The following example uses exceptions, which are discussed in Exception Mechanism.

```
// Sample.zpp

#include<iostream.h>
#include<exception.h>

using namespace ioSpace;
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
// userExceptionEvents defines new exceptions by extending
// system exceptionEventType.
// ------------------------------------------------------------------ //

enum userExceptionEvents : exceptionEventType {
      _FirstUserException,
      _SecondUserException,
      _ThirdUserException
};

////////////////////////////////////////////////////////////////////////
// This will be used as base for deriving next class.
// It has two invariants.
// ------------------------------------------------------------------ //
```

```
class baseInvariantClass
      double d;
      short s;

      invariant (d > 97.47) _FirstUserException;
      invariant (s == 7) _SecondUserException;

public:

      baseInvariantClass(short);
      void changeMember(double);
end;

baseInvariantClass::baseInvariantClass(short i)
      s = i;
      d = 100;
end;

void baseInvariantClass::changeMember(double b)
      d = b;
end;

////////////////////////////////////////////////////////////////////
// This class has two invariants, and inherits the invariants
// of baseInvariantClass, as well.
// ------------------------------------------------------------- //

class derivedInvariantClass : baseInvariantClass

      invariant (a < b) trigger();
      invariant (a > 25) _ThirdUserException;

      int a, b;

      void trigger(void);

public:

      derivedInvariantClass(short);
      void Checker(void);
end;

derivedInvariantClass::derivedInvariantClass(short i)
: baseInvariantClass(i)

      a = 50;
      b = 40;
end;

void derivedInvariantClass::Checker(void)
      a = 30;
      b = 40;
      changeMember(90.20);
end;

void derivedInvariantClass::trigger(void)
      output << "derivedInvariantClass::trigger(void)\n";
```

```
end;

////////////////////////////////////////////////////////////////////////
// The type plain, has no members and no invariants.
// However, it inherits all invariants of its base.
// ------------------------------------------------------------------ //

struct plain : derivedInvariantClass

      plain(short);

end;

plain::plain(short i) : derivedInvariantClass(i)
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      layer<userExceptionEvents>

// This will invoke trigger() twice. First in constructor
// for derivedInvariantClass. and because the trigger()
// does not repair the problem, it is invoked again in the
// constructor for the type plain itself, due to inheriting
// invariants.

            plain p(7);

// Checker() sets members a and b, satisfying invariant (a < b)
// so trigger() is not invoked.
// However, it calls changeMember() which sets the member d of class
// baseInvariantClass so that the invariant (d > 97.47)
// is violated. This violation raises _FirstUserException.

            p.Checker();

      handler

            case _FirstUserException:
                  output << "Caught _FirstUserException\n";

            case _SecondUserException:
                  output << "Caught _SecondUserException\n";

            case _ThirdUserException:
                  output << "Caught _ThirdUserException\n";

      endlayer;

      output << "Good-bye World!\n";
```

```
end;
```

The output of this program is as follows.

```
Hello World!
derivedInvariantClass::trigger(void)
derivedInvariantClass::trigger(void)
Caught _FirstUserException
Good-bye World!
```

Section III.2.J. Method Constraints

Constraints are conditions that are expected to be true prior to the execution of a **public** method. Constraints for a method are **boolean** expressions that **must involve the formal parameters of the method**, and may also involve the members of the **class** to which the method belongs. A **public** method can have any number of constraints.

Constraints are not associated with inheritance. The re-definition of a method in a derived **class** does **not inherit its constraints**. The new definition of the method must provide its own constraints if it needs to do so. However, it may also call the method that it is re-defining, in which case the constraints of the called method will be tested.

Constraints for a method are specified at its prototype within a **class** definition, between a pair of {}. A constraint is a **boolean** expression followed by one of two possible courses of action in case the constraint is violated: raising an exception or invoking a **private** method (called a **trigger**) to rectify the problem.

After completion of its execution, a trigger returns to its caller unless its execution raises an exception. Thus, **if the trigger actually rectifies the problem the execution of the method that triggered it will proceed as usual**.

The following example uses exceptions, which are discussed in Exception Mechanism.

```
// Example.zpp

#include<iostream.h>
#include<exception.h>

using namespace ioSpace;
using namespace exceptionSpace;

//////////////////////////////////////////////////////////////////////
// userExceptionEvents defines new exceptions by extending
// system exceptionEventType.
// ---------------------------------------------------------------- //

enum userExceptionEvents : exceptionEventType {
      _FirstUserException,
      _SecondUserException,
      _ThirdUserException
```

```
};

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //

class constraintType
      int i;
      double d;

      void trigger(double);   //constraint trigger method

public:

      constraintType(void);

      void constrainedMethodInt(int j)
      {
            (j < i) _FirstUserException;
            (j > d) _SecondUserException;
      };

      void constrainedMethodDouble(double b)
      {
            (b != i) _ThirdUserException;
            (b >= d) trigger(b);
      };

end;

// ------------------------------------------------------------------ //

constraintType::constraintType(void)
      i = 170;
      d = 97.47;
end;

void constraintType::constrainedMethodInt(int j)
end;

void constraintType::constrainedMethodDouble(double b)
end;

void constraintType::trigger(double b)
      output << "constraintType::trigger() " << b << '\n';
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      layer<userExceptionEvents>

            constraintType constraintObject;
```

```
// This call will invoke trigger.

            constraintObject.constrainedMethodDouble(13.5);

// This will raise _SecondUserException.

            constraintObject.constrainedMethodInt(90);

      handler

            case _FirstUserException:
                  output << "Caught _FirstUserException\n";

            case _SecondUserException:
                  output << "Caught _SecondUserException\n";

            case _ThirdUserException:
                  output << "Caught _ThirdUserException\n";

      endlayer;

      output << "Good-bye World!\n";

end;
```

The output of this program is as follows.

```
Hello World!
constraintType::trigger() 13.5
Caught _SecondUserException
Good-bye World!
```

Section III.2.K. Common Members

A **common** member, as the name indicates, is common among all instances of a **class**. That means, if any instance of a **class** modifies a **common** member, all instances of the **class** will receive the change.

A **common** member must be initialized generally right after the definition of the **class** that owns it. After initialization, only **authorized** methods can modify a **common** member.

The list of authorized methods is a comma-separated list of methods of a **class** that are authorized to modify a **common** member. Each method is listed in the form of its prototype without its return.

A **common** member cannot be a **public** member.

Instances created after changing a **common** member will see the latest value of the **common** member. Thus, at any time all instances will have same value for their **common** member.

A **common** member cannot be modified until an instance of its **class** is created. Authorized methods can only be invoked on instances of a **class**. Thus, unlike C++ there are no static methods to modify a static member prior to construction of any instance of a **class**. Static methods and members essentially **turn a class into a namespace**.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////

struct TestType

      int i;

protected:

// setCommon() is authorized to modify g

      common long g : setCommon(long);

public:

      TestType(int);

      void setCommon(long);
      long getCommon(void);

end;

// ------------------------------------------------------------------ //
// A common member must be initialized before instances
// can be created.

long TestType::g = 777;

// ------------------------------------------------------------------ //

TestType::TestType(int n)
      i = n;
end;

void TestType::setCommon(long l)
      g = l;
end;

long TestType::getCommon(void)
      return g;
end;
```

```
////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //

entry void main(void)

      TestType tt_1(55);
      TestType tt_2(77);
      TestType tt_3(99);

      output << "Initial value of common member for tt_1 is: " <<
          tt_1.getCommon() << '\n';
      output << "Initial value of common member for tt_2 is: " <<
          tt_2.getCommon() << '\n';
      output << "Initial value of common member for tt_3 is: " <<
          tt_3.getCommon() << '\n';

      tt_2.setCommon(12345);

      TestType tt_4(11);

      output <<  "Now, value of common member for tt_1 is: " <<
          tt_1.getCommon() << '\n';
      output <<  "Now, value of common member for tt_2 is: " <<
          tt_2.getCommon() << '\n';
      output <<  "Now, value of common member for tt_3 is: " <<
          tt_3.getCommon() << '\n';
      output <<  "Now, value of common member for tt_4 is: " <<
          tt_4.getCommon() << '\n';

end;
```

The output of this program is as follows.

```
Initial value of common member for tt_1 is: 777
Initial value of common member for tt_2 is: 777
Initial value of common member for tt_3 is: 777
Now, value of common member for tt_1 is: 12345
Now, value of common member for tt_2 is: 12345
Now, value of common member for tt_3 is: 12345
Now, value of common member for tt_4 is: 12345
```

Important Remark. An authorized method can modify a **common** member in such a way that invariants of the instance on which the method is called are not violated, and yet the invariants are violated for many other instances. It is not practical to check the invariants for all instance, nor there is a convenient way of reporting multiple violations. For that reason, **invariant** expressions cannot involve **common** members. Instead, use **constraints** for **authorized method** to ensure the correct state for **common** members.

Section III.2.L. Friendship

Specifying a global function as **friend** of a **class** grants the function the privilege of accessing none **public** members of the **class**. Since a **friend** can modify none **public** members, this is clearly not a good idea as it may result in obscure defects. Below is an example of bad use of notion of **friend**.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////

class Simple
      int i;
      double d;

public:

      Simple(void);
      friend outStream& operator<<(outStream&, const Simple&);

end;

Simple::Simple(void)
      i = 777;
      d = 47.97;
end;

outStream& operator<<(outStream& out, const Simple& e)
      out << e.i << '\n';
      out << e.d << '\n';
      return out;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //

entry void main(void)
      output << "Hello World!\n";

      Simple s;
      output << s;

      output << "Good-bye World!\n";
end;
```

Instead, use the **visibility** specification for members that you intend to output as shown in the following example.

```
// Sample.zpp
```

```
#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////

class Simple
      int i <visible>;
      double d <visible>;

public:

      Simple(void);
end;

Simple::Simple(void)
      i = 777;
      d = 47.97;
end;

outStream& operator<<(outStream& out, const Simple& e)
      out << e.i << '\n';
      out << e.d << '\n';
      return out;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //

entry void main(void)
      output << "Hello World!\n";

      Simple s;
      output << s;

      output << "Good-bye World!\n";
end;
```

The output of both examples is the same.

```
Hello World!
777
47.97
Good-bye World!
```

However, there are times that allowing, one **class** to access none **public** members of another **class** is, convenient and safe. For instance, in templates the **friend class** Iterator is allowed to modify the member of its List, at which it is pointing. Any other solution for modifying a member of the list during iteration will be more complex.

It is recommended to use the **friend** mechanism only when one anticipates modification to none **public** members of a **class** that will result in **excessive irrelevant coding** than using the specification **visible**.

Section III.2.M. Explicit Cast Specification

Compiler uses two kinds of methods of structural types in its implicit conversions. When attempting to promote the type of an object, the compiler looks for a **constructor**. On the other hand, when attempting to reduce the type of object, the compiler looks for a **conversion operator**.

The **cast** specification for constructors and conversion operators **prohibits** the compiler from using them **implicitly**. User must **explicitly** instruct the compiler to use a method specified as **cast**. The operator for **explicit cast** is also **cast**, as illustrated by the example. For use of **cast** operator with regard to inheritance see Inheritance and Casting.

The **cast** operator takes two arguments. The first argument is the **type** to cast to, and the second argument is the object to cast. Cast creates an object of type of its first argument from the object given to it for its second argument. The **cast** operator returns (evaluates to) the object it creates. Below is a simple use of **cast** explaining how it is used in the example.

```
double dbl = 3.4;
int integer = cast(int, dbl); // integer == 3
```

Below is the example for this section. Note that two conversion operators, and one constructor are specified **cast**. In the example, wherever **explicit cast** is required has been commented. That means, if you do not use explicit **cast** the compiler will report that it could not find a suitable conversion.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

struct firstType
      double d;

      firstType(void);
      operator double(void) cast;
end;

firstType::firstType(void)
      d = 4.7;
end;

firstType::operator double(void)
      return d;
end;

//-------------------------------------------------------------------//
```

```
struct secondType
      double b;

      secondType(const firstType&) cast;
      operator double(void) cast;
end;

secondType::secondType(const firstType& f) : b(f.d)
end;

secondType::operator double(void)
      return b;
end;

//-----------------------------------------------------------------//

double multiply(double d, secondType st)
      return d * cast(double, st); // Explicit cast required
end;

////////////////////////////////////////////////////////////////////
//----------------------------------------------------------------//
////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      firstType ft;
      double dbl = 5.3;

// Explicit cast is required in next statement.

      double dblInstance = dbl * cast(double, ft);
      output << dblInstance << '\n';

// Explicit constructor cast secondType(ft) is required

      dblInstance = multiply(dbl, secondType(ft));
      output << dblInstance << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
24.91
24.91
Good-bye World!
```

Section III.2.N. Const Specification

A **method** specified **const** is not allowed to change members of its **class**. The **const** specification avoids coding errors that may occur, especially when making changes to a large method.

Specifying a pass-by-reference parameter as **const** protects the object passed to the call. Specifying a method as **const** protects the object on which the method is invoked.

A conversion operator can be specified **const** and **cast**. That is, the two specifications for a method are independent. However, **const** must come before **cast**, as shown below.

```
struct firstType
      double d;

      firstType(void);
      operator double(void) const cast;
end;

firstType::firstType(void)
      d = 4.7;
end;

firstType::operator double(void)
      return d;
end;
```

Note that specifications are only needed at **prototype** of a method, and must not be repeated at the definition of the method.

Section III.2.O. Polymorphism

One view of pointer in programming is that a pointer is a variable that ranges over a set of objects of a **specific type**. For instance, a pointer to **double**, i.e. and instance of `double*` can traverse a list of objects of type **double**.

An algebraic identity is valid for different kinds of numbers: whole numbers, integers, rational numbers, and so on. In this case, the mathematical notion of variable implies ranging over different types of numbers. Polymorphism extends the variable view of a pointer to the latter mathematical view. A pointer to the base type of an inheritance hierarchy can point to any instance of a type in the hierarchy.

In the example, we start with a base type `animal`. In one direction we derive the types `dogFamily`, `dog` and `wolf`. In another direction we derive `catFamily`, `tiger` and `lion`. The pointer `animal* Animal` (member of **`struct`** `link`) will be used to point to instances of all these types. This allows us to insert instances of all of these types into a list. As a result, in the `main()` function, we are able to make a short iterative statement and print the names of all animals in the list.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

//---------------------------------------------------------------------//
// This will be used for base type of inheritance.

class animal

protected:

      string name;

public:

      animal(void);
      string Kind(void);
end;

animal::animal(void)
      name = "Animal";
end;

string animal::Kind(void)
      return name;
end;

//---------------------------------------------------------------------//
// In one direction we derive the dog family.
```

```
class dogFamily : animal

public:

      dogFamily(void);
      string Kind(void);
end;

dogFamily::dogFamily(void)
      name = "Dog Family";
end;

string dogFamily::Kind(void)
      return name;
end;

//-------------------------------------------------------------------//

class dog : dogFamily

public:

      dog(void);
      string Kind(void);
end;

dog::dog(void)
      name = "Dog";
end;

string dog::Kind(void)
      return name;
end;

//-------------------------------------------------------------------//

class wolf : dogFamily

public:

      wolf(void);
      string Kind(void);
end;

wolf::wolf(void)
      name = "Wolf";
end;

string wolf::Kind(void)
      return name;
end;

//-------------------------------------------------------------------//
// In another direction we derive the cat family.

class catFamily : animal
```

```
public:

      catFamily(void);
      string Kind(void);
end;

catFamily::catFamily(void)
      name = "Cat Family";
end;

string catFamily::Kind(void)
      return name;
end;

//---------------------------------------------------------------------//

class tiger : catFamily

public:

      tiger(void);
      string Kind(void);
end;

tiger::tiger(void)
      name = "Tiger";
end;

string tiger::Kind(void)
      return name;
end;

//---------------------------------------------------------------------//

class lion : catFamily

public:

      lion(void);
      string Kind(void);
end;

lion::lion(void)
      name = "Lion";
end;

string lion::Kind(void)
      return name;
end;

//---------------------------------------------------------------------//

struct link
      animal* Animal;
      link* next;

      link(void);
```

```
      link(animal*);
end;

link::link(void)
      Animal = 0;
      next = 0;
end;

link::link(animal* a)
      Animal = a;
      next = 0;
end;

//-----------------------------------------------------------------------//

struct animalList
      link* last;

      animalList(void);
      void append(animal*);
      void clear(void);
end;

animalList::animalList(void)
      last = 0;
end;

void animalList::append(animal* a)
      if (last)
            link* lk = new link(a);
            lk->next = last->next;
            last->next = lk;
      else
            last = new link(a);
            last->next = last;
      endif;
end;

void animalList::clear(void)
      while (last)
            if (last == last->next)
                  delete last;
                  last = 0;
            else
                  link* lk = last->next;
                  last->next = lk->next;
                  delete lk;
            endif;
      endwhile;
end;

////////////////////////////////////////////////////////////////////////
//-----------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";
```

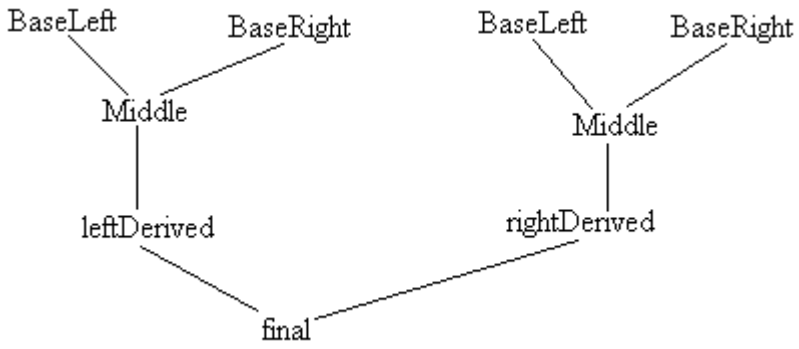
```
//---------------------------------------------------------------------//
// The arguments to append are of different types. But
// they are all derived from class animal. So, a pointer
// to the base type animal (of struct link) can point to
// any one of them.

      animalList aList;
      aList.append(new dog);
      aList.append(new tiger);
      aList.append(new wolf);
      aList.append(new lion);

//---------------------------------------------------------------------//
// Initialize pointers before entering the do loop.
// In the loop, current is a polymorphic pointer. and
// invokes the method Kind() of the instance it is pointing to.

      link* start = aList.last;

      if (start)
        link* current = start;

          do
            output << "Animal is " << current->Animal->Kind() << '\n';
            current = current->next;
          enddo (start == current);
      endif;

//---------------------------------------------------------------------//
// Empty the list.

      aList.clear();

//---------------------------------------------------------------------//

      output << "Good-bye World!\n";
end;
```

The output is as follows.

```
Hello World!
Animal is Dog
Animal is Lion
Animal is Wolf
Animal is Tiger
Good-bye World!
```

Section III.2.P. Inheritance and Casting

One of the functions of **cast** operator is moving up and down an inheritance hierarchy. In this section, by moving up we mean moving up to reach a base. We move down to reach a derived type. This usage is for the following figure, which is the inheritance hierarchy of the example we shall use to illustrate the action of **cast**.



We use code fragments from the complete program listed below to explain the actions of **cast** operator.

Cast is safe when moving up (casting up to a base type). Consider the following.

```
rightDerived* rd = new rightDerived();
BaseLeft* bl = cast(BaseLeft*, rd);
```

In the above example, starting from type rightDerived, **cast** will find BaseLeft via depth-first search. It will then create a pointer of type BaseLeft, making it point to the base of object rd, which is assigned to the pointer bl.

However, moving down (casting down to a derived type) is not as safe and needs care. Consider the following, as next line after the two lines of code above.

```
leftDerived* ld = cast(leftDerived*, bl);
```

In this case, **cast** is told to move down from BaseLeft to leftDerived. As shown in the figure, cast will succeed, with wrong result. The problem is that the object bl is a base for the object rd of type rightDerived. However, at compile time we are not dealing with actual objects, only relationships among their types.

The last thing to remember is that **cast** can only go up or down. To make it go both ways you need to use **cast** for each direction.

```
cast(BaseRight*, cast(rightDerived*, cast(final*, fbl)));
```

In the above line, from inside out, this is what happens.

We start with fbl of type BaseLeft. The object fbl is a base of leftDerived, the **top-left** of the figure. We wish to move to the base BaseRight, **top-right** of the figure.

The first **cast** takes us all the way down to the type final. Because of depth-first search, we first move up to rightDerived, and finally the last **cast** takes us safely to BaseRight at **top-right** of the figure.

If **cast** operator does not find the requested type in either moving up or else moving down, it will simply act as a **C-pointer cast**. In the example this is illustrated by casting from BaseLeft to BaseRight. Object fbl is of type BaseLeft.

```
BaseRight* fbr = cast(BaseRight*, fbl);
```

Since neither type is a base for the other, the above line results in a plain C-pointer casting.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

struct BaseLeft
      int i;

      BaseLeft(void);
end;

BaseLeft::BaseLeft(void)
      i = 777;
end;

// ------------------------------------------------------------------ //

struct BaseRight
      short s;

      BaseRight(void);
end;

BaseRight::BaseRight(void)
      s = 47;
end;

// ------------------------------------------------------------------ //

struct Middle : BaseLeft, BaseRight
      double d;

      Middle(void);
```

```
end;

Middle::Middle(void)
	d = 5.5;
end;

// ------------------------------------------------------------------ //

struct leftDerived : Middle
	double d;

	leftDerived(void);
end;

leftDerived::leftDerived(void)
	d = 1.1;
end;

// ------------------------------------------------------------------ //

struct rightDerived : Middle
	double d;

	rightDerived(void);
end;

rightDerived::rightDerived(void)
	d = 2.2;
end;

// ------------------------------------------------------------------ //

struct final : leftDerived, rightDerived
	int n;

	final(void);
end;

final::final(void)
	n = 347;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

	rightDerived* rd = new rightDerived();
	output << rd->d << '\n';

	BaseLeft* bl = cast(BaseLeft*, rd); // OK
	output << bl->i << '\n';

// cast yields incorrect pointer

	leftDerived* ld = cast(leftDerived*, bl);
```

```
      output << ld->d << '\n';

      delete rd;

// ------------------------------------------------------------------ //

      final* fp = new final();

      BaseLeft* fbl = cast(BaseLeft*, fp); // OK
      output << fbl->i << '\n';

// Plain C-pointer cast with wrong result

      BaseRight* fbr = cast(BaseRight*, fbl);
      output << fbr->s << '\n';

// Operator ->> is similar to -> for reaching a base.

      fp->>rightDerived::Middle::BaseRight::s = 99;

// Correct recursive use of cast.

      fbr = cast(BaseRight*, cast(rightDerived*, cast(final*, fbl)));
      output << fbr->s << '\n';

      delete fp;

end;
```

The output is as follows.

```
2.2
777
2.2
777
777
99
```

Chapter III.3. Task

The type constructor **task** has all the characteristics of **class,** and in addition to those, it has its own characteristics that we are going to discuss in this section.

One important characteristic of **task** is that an instance of a **task** is **threaded**. That is, at creation, first a thread is created and then the instance of **task** is created in the new thread.

A call to a **public** method of **task** **blocks** until the **task** object **services** the call. A **task** object has a **queue** and responds to calls made to its **public** methods in the order they arrive.

Members of type **task** of a **task** will have their own thread. This is true of **task** **bases** from which a **task** is derived. Thus, an instance of **task** may initiate **multiple threads** at its construction.

The thread of a **task** instance is created at its construction and ends at the destruction of the instance. Ordinarily, the duration of the execution of the thread is from the declaration of a **task** object until it goes out of scope. However, this can be extended by, creating dynamic tasks via the **new** operator. **In this case the thread is terminated when the instance is deleted**.

As **task** object must be passed to a call by reference, or as a pointer. Compiler makes **private** and **empty** copy constructor and default assignment operator for a **task**, preventing their use.

It makes no sense to compare two **task** objects, which are threads with their own queue of requests for service, etc. Therefore, compiler also makes the default operators for comparison **private**.

It should be noted that a **class** that either derives from a **task**, or includes a **task** as a member, is treated same as **task**. Such a **class** will have its copy constructor, assignment operator and comparison operators, **private**.

The following example illustrates these preliminary characteristics of **task** type constructor.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

//////////////////////////////////////////////////////////////////////
// One will be used as member of Two.
// ------------------------------------------------------------- //
```

```
task One
	int i;
public:
	One(void);
	~One(void);

	void Show(void);
	void Set(void);
	void Set(int);
end;

One::One(void)
	i = 5;
end;

One::~One(void)
	output << "Inside One::~One(void).\n";
end;

void One::Show(void)
	output << i << '\n';
end;

void One::Set(void)
	i += 7;
end;

void One::Set(int k)
	i = k;
end;

// ------------------------------------------------------------------ //
// Two has a task member (One), and will be used as base for Three.

task Two
	double d;
	One n;

public:

	Two(void);
	~Two(void);
	void Set(double);
	void Show(void);
	double get(void);

	void showOne(void);
end;

Two::Two(void)
	d = 3.7;
end;

Two::~Two(void)
	output << "Inside Two::~Two(void).\n";
end;
```

```
void Two::Set(double b)
      d = b;
end;

double Two::get(void)
      return d;
end;

void Two::Show(void)
      output << d << '\n';
end;

void Two::showOne(void)
      n.Show();
end;

////////////////////////////////////////////////////////////////////////
// Three, is derived from another task, which has a task member (One).
// ------------------------------------------------------------------ //

task Three : Two
      long l;
public:
      Three(long);
      ~Three(void);
      long give(void);
      void Show(void);
end;

Three::Three(long g)
      l = g;
end;

Three::~Three(void)
      output << "Inside Three::~Three(void).\n";
end;

long Three::give(void)
      return l;
end;

void Three::Show(void)
      Two::Show();
      Two::showOne();
end;

////////////////////////////////////////////////////////////////////////
// Instances of task must be passed by reference, or pointer.
// ---------------------------------------------------------------- //
// The following are two examples of passing tasks to global functions.
////////////////////////////////////////////////////////////////////////

Three& global(Three& h)
      output << "Start of global\n";
      h.Show();
      output << "End of global\n";
      return h;
```

```
end;

Three* ptrGlobal(Three* p)
      output << "Start of ptrGlobal\n";
      p->Show();
      output << "End of ptrGlobal\n";
      return p;
end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Note that instance t will have three threads,
// one for its base (Two), and one
// for member (One) of its base, in addition to itself.
// Also, the threads of Three are destroyed after function
// main() ends.

      Three t(77);
      t.Show();
      output << t.give() << '\n';

      global(t).Show();

// The threads of tp are created here, and destroyed
// at delete.

      Three* tp = new Three(99);
      ptrGlobal(tp)->Show();
      delete tp;

      output << "Good-bye World!\n";
end;
```

The output of both examples is the same.

```
Hello World!
3.7
5
77
Start of global
3.7
5
End of global
3.7
5
Start of ptrGlobal
3.7
5
End of ptrGlobal
3.7
```

```
5
Inside Three::~Three(void).
Inside Two::~Two(void).
Inside One::~One(void).
Good-bye World!
Inside Three::~Three(void).
Inside Two::~Two(void).
Inside One::~One(void).
```

Section III.3.A. Task Idler

An instance of a **task** runs in its own thread, checking for arriving requests for its **public** methods and servicing those requests. When there are no requests to service, a **task** object can do other things. In this subsection we look at **task** idler, and in the next subsection we discuss **task** signal handlers. The illustrating example is given in next subsection.

A **task** idler is a **private** method, which **cannot be called directly**. Instead, **task** object invokes its **idler** when there is nothing else to do, thus the name idler.

The idler method of a **task** has the same name as **task**, prefixed with the symbol @. The signature of idler is **void**.

The compiler generates an empty idler for a **task**, unless you provide one. Each **task** has its own idler. Thus, **the idler is not inherited**. When deriving a **task** from another, unless you define an idler for the derived **task** it will receive an **empty idler** from the compiler, whether or not the base's idler is empty.

Section III.3.B. Task Signal Handler

For the notion of signaling please refer to Signaling.

Task objects can respond to signals just as they service incoming requests for their **public** methods. The method of a **task** responding to a signal is called a **handler**.

A **task** signal handler is a **private** method that cannot be invoked directly. Instead, **task** object invokes the appropriate handler when it receives a signal.

The return and signature of a handler are **void**. A handler is attached to the signal it must respond to at its prototype, as shown below.

```
void ATaskHandler(void)<_SIGNAL_SomeSignal>;
```

Handlers are not inherited. However, remember that the base **task** from which a **task** is derived runs in its own thread and **will respond to signals**. This is trivially true of members of a **task** that are **tasks with handlers**.

The following example illustrates the notions of **idler** and signal handlers.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
// Define new signals by extending system signals.
// ------------------------------------------------------------------ //

enum MyServiceSignals : signalEventType {
      _SIGNAL_FirstIdler,
      _SIGNAL_SecondIdler,
      _SIGNAL_ThirdIdler
};

////////////////////////////////////////////////////////////////////////

task idler

// Private methods

      @idler(void); // the task idler

// Three signal handlers, one for each signal.

      void FirstHandler(void)<_SIGNAL_FirstIdler>;
      void SecondHandler(void)<_SIGNAL_SecondIdler>;
      void ThirdHandler(void)<_SIGNAL_ThirdIdler>;

public: // no public methods for this example
end;

// ------------------------------------------------------------------ //
// Idler definition

idler::@idler(void)
      output << "Inside idler...\n";
end;

// ------------------------------------------------------------------ //
// Definitions for handlers

void idler::FirstHandler(void)
      output << "Inside FirstHandler...\n";
end;
```

```
void idler::SecondHandler(void)
      output << "Inside SecondHandler...\n";
end;

void idler::ThirdHandler(void)
      output << "Inside ThirdHandler...\n";
end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      idler idl; // Create an instance of task

// Generate all signals

      signal <- _SIGNAL_FirstIdler;
      signal <- _SIGNAL_SecondIdler;
      signal <- _SIGNAL_ThirdIdler;

// Following loop is not needed. However, it gives a chance
// to task object to run a few time slices before this
// main thread ends, terminating execution.

      for (int i = 0; i < 10; i++) endfor;

      output << "Good-bye World!\n";
end;
```

The output of both examples is the same.

```
Hello World!
Inside FirstHandler...
Inside SecondHandler...
Inside ThirdHandler...
Inside idler...
Inside idler...
Inside idler...
Inside idler...
Inside idler...
Inside idler...
Inside idler...
Inside idler...
Inside idler...
Good-bye World!
```

Section III.3.C. Task Accept Signals

This section illustrates the use of process-bounded entire signals with regard to tasks. These signals, when generated, will be caught by all threads that have **registered** to catch them. The registration is done via the **accepts** list.

For **task** type constructor the **accepts** list specification, for registering entire signals to catch, can appear anywhere within its definition. However, for readability it is better to put **accepts** lists around the top of the definition, like the **invariant** specifications. Handlers for entire signals retain their syntax and semantics as with the simple kind of signals illustrated in previous subsection.

In the following example, two **task** types are defined, One and Two. Both types register to accept the same entire signals. In main thread, an instance of each type is created. Then, an entire signal is generated and the main thread waits until both **task** objects respond by generating reply signals. The main thread then generates the second entire signal and waits for response from **task** objects, before terminating.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
// Define plain and entire signals by extending
// corresponding system signals.
// ------------------------------------------------------------------ //

enum MyFirstSignals : signalEventType {
      _SIGNAL_FirstSignal,
      _SIGNAL_SecondSignal,
      _SIGNAL_ThirdSignal,
      _SIGNAL_OneSignal,
      _SIGNAL_TwoSignal,
      _SIGNAL_ThreeSignal
};

enum MyFirstThreadSignals : threadEntireEventType {
      _THREAD_ENTIRE_FirstSignal,
      _THREAD_ENTIRE_SecondSignal,
      _THREAD_ENTIRE_ThirdSignal
};

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //

task One

// Register the entire signals to accept
```

```
      accepts(_THREAD_ENTIRE_FirstSignal, _THREAD_ENTIRE_SecondSignal);

      @One(void); // idler

// Handlers for accepts list of entire signals

      void FirstHandlerOne(void)<_THREAD_ENTIRE_FirstSignal>;
      void SecondHandlerOne(void)<_THREAD_ENTIRE_SecondSignal>;

public:
end;

// ------------------------------------------------------------------ //

One::@One(void)
      output << "Inside One::@One(void)\n";
end;

void One::FirstHandlerOne(void)
      output << "Inside FirstHandlerOne(void). ";
      output << "Sending signal _SIGNAL_FirstSignal.\n";
      signal <- _SIGNAL_FirstSignal;
end;

void One::SecondHandlerOne(void)
      output << "Inside SecondHandlerOne(void). ";
      output << "Sending signal _SIGNAL_SecondSignal.\n";
      signal <- _SIGNAL_SecondSignal;
end;

//////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //

task Two

// Register the entire signals to accept

      accepts(_THREAD_ENTIRE_FirstSignal, _THREAD_ENTIRE_SecondSignal);

      @Two(void); // idler

// Handlers for accepts list of entire signals

      void FirstHandlerTwo(void)<_THREAD_ENTIRE_FirstSignal>;
      void SecondHandlerTwo(void)<_THREAD_ENTIRE_SecondSignal>;

public:
end;

// ------------------------------------------------------------------ //

Two::@Two(void)
      output << "Inside Two::@Two(void)\n";
end;

void Two::FirstHandlerTwo(void)
```

```
      output << "Inside FirstHandlerTwo(void). ";
      output << "Sending signal _SIGNAL_OneSignal.\n";
      signal <- _SIGNAL_OneSignal;
end;

void Two::SecondHandlerTwo(void)
      output << "Inside SecondHandlerTwo(void). ";
      output << "Sending signal _SIGNAL_TwoSignal.\n";
      signal <- _SIGNAL_TwoSignal;
end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

// ---------------------------------------------------------------- //
// Create task objects

      One N;
      Two T;

// ---------------------------------------------------------------- //
// Generate an entire signal

      output << "Main: sending entire signal ";
      output << "_THREAD_ENTIRE_FirstSignal\n";
      signal <- _THREAD_ENTIRE_FirstSignal;

// ---------------------------------------------------------------- //
// Wait until both task objects respond

      output << "Main: Waiting for _SIGNAL_FirstSignal....\n";
      do enddo(signal ? _SIGNAL_FirstSignal);

      output << "Main: Waiting for _SIGNAL_OneSignal....\n";
      do enddo(signal ? _SIGNAL_OneSignal);

// ---------------------------------------------------------------- //
// Repeat the above pattern for second entire signal.
// ---------------------------------------------------------------- //

      output << "Main: sending entire signal ";
      output << "_THREAD_ENTIRE_SecondSignal\n";
      signal <- _THREAD_ENTIRE_SecondSignal;

      output << "Main: Waiting for _SIGNAL_SecondSignal....\n";
      do enddo(signal ? _SIGNAL_SecondSignal);

      output << "Main: Waiting for _SIGNAL_TwoSignal....\n";
      do enddo(signal ? _SIGNAL_TwoSignal);

// ---------------------------------------------------------------- //

      output << "Good-bye World!\n";
```

```
end;
```

The output of this example is as follows.

```
Hello World!
Inside One::@One(void)
Inside Two::@Two(void)
Main: sending entire signal _THREAD_ENTIRE_FirstSignal
Main: Waiting for _SIGNAL_FirstSignal....
Inside One::@One(void)
Inside Two::@Two(void)
Inside FirstHandlerOne(void). Sending signal _SIGNAL_FirstSignal.
Inside One::@One(void)
Inside FirstHandlerTwo(void). Sending signal _SIGNAL_OneSignal.
Inside Two::@Two(void)
Main: Waiting for _SIGNAL_OneSignal....
Inside One::@One(void)
Inside Two::@Two(void)
Main: sending entire signal _THREAD_ENTIRE_SecondSignal
Main: Waiting for _SIGNAL_SecondSignal....
Inside SecondHandlerOne(void). Sending signal _SIGNAL_SecondSignal.
Inside SecondHandlerTwo(void). Sending signal _SIGNAL_TwoSignal.
Main: Waiting for _SIGNAL_TwoSignal....
Inside One::@One(void)
Inside Two::@Two(void)
Good-bye World!
Inside One::@One(void)
Inside Two::@Two(void)
```

Section III.3.D. Task Hear Signals

Tell/Hear signals are discussed in chapter on Distributed Signaling. In the example presented here we use tell/hear signals for communication of data between the main thread, and a local **task** thread created in the main thread. This is for the sake of keeping the example simple. The syntax and semantics remain the same as for a distributed case.

The main thread sends two **tell** signals to the **task** object, and in each case waits for a reply. In this example the reply is a simple signal. **In a distributed case the response would have been a tell/hear signal**.

The second **tell** signal sent by the main thread does not carry any data. For a local use this reduces a tell/hear signal to an entire signal illustrated in previous subsection. However, in a distributed case we could only use tell/hear signals.

For readability, the **hears** specifications for a **task** should come near the top of the definition of the type, like the **accepts** specification. The handler prototypes for **hears** signals include the members that are intended to receive the data arriving from a **tell** signal. Otherwise, the syntax and semantics of handlers remain the same.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
// Make new signals by extending corresponding system signals.
// ------------------------------------------------------------------ //

enum MyFirstSignals : signalEventType {
      _SIGNAL_OneTellSignal,
      _SIGNAL_TwoTellSignal
};

enum MyTellSignals : tellSignalType {
      _TELL_telling_1,
      _TELL_telling_2,
      _TELL_telling_3
};

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //

task One

      int ti;
      double di;
```

```
// Register tell/hear signals.

      hears(_TELL_telling_1<int, double>, _TELL_telling_2);

      @One(void); // idler

// Handler for tell/hear signals.

      void FirstHearHandlerOne(void)<_TELL_telling_1 : ti, di>;
      void SecondHearHandlerOne(void)<_TELL_telling_2>;

public:

      One(void);

end;

// ------------------------------------------------------------------ //

One::One(void)
      ti = 3;
      di = 5.2;
end;

One::@One(void)
      output << "Inside One::@One(void): " << ti << ' ' << di << '\n';
end;

void One::FirstHearHandlerOne(void)
      output << "Inside FirstHearHandlerOne(void). ";
      output << "Sending signal _SIGNAL_OneTellSignal.\n";
      output << "FirstHearHandlerOne(void), got: " << ti
         << ' ' << di << '\n';
      signal <- _SIGNAL_OneTellSignal;
end;

void One::SecondHearHandlerOne(void)
      output << "Inside SecondHearHandlerOne(void). ";
      output << "Sending signal _SIGNAL_TwoTellSignal.\n";
      signal <- _SIGNAL_TwoTellSignal;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

// ------------------------------------------------------------------ //
// Create a task object

      One N;

// ------------------------------------------------------------------ //
// Make some data and tell them to task object
```

```
      int mm = 25;
      double dd = 56.12;

      output << "Main: sending tell signal _TELL_telling_1\n";
      tell <- _TELL_telling_1 : mm, dd;

// ------------------------------------------------------------------ //
// Wait until task object replies

      output << "Main: Waiting for _SIGNAL_OneTellSignal....\n";
      do enddo(signal ? _SIGNAL_OneTellSignal);

// ------------------------------------------------------------------ //
// Repeat above pattern with second tell signal.
// This tell signal does not send any data. It
// this context it behaves like an entire signal.
// However, note that tell/hear signals are intended
// for distributed communication.
// ------------------------------------------------------------------ //

      output << "Main: sending tell signal _TELL_telling_2\n";
      tell <- _TELL_telling_2;

      output << "Main: Waiting for _SIGNAL_TwoTellSignal....\n";
      do enddo(signal ? _SIGNAL_TwoTellSignal);

      output << "Good-bye World!\n";
end;
```

Remark. The example has two threads. The outputs of threads intervene.

The output of this example is as follows.

```
Hello World!
Main: sending tell signal _TELL_telling_1
Inside One::@One(void): 3Main: Waiting for _SIGNAL_OneTellSignal....
 5.2
Inside FirstHearHandlerOne(void). Sending signal _SIGNAL_OneTellSignal.
FirstHearHandlerOne(void), got: 25 56.12
Main: sending tell signal _TELL_telling_2
Inside One::@One(void): 25Main: Waiting for _SIGNAL_TwoTellSignal....
 56.12
Inside SecondHearHandlerOne(void).
  Sending signal _SIGNAL_TwoTellSignal.
Good-bye World!
```

Section III.3.E. Arrays of Tasks

An array of tasks, regular or dynamic array, creates all **task** instances at once. Each **task** instance is created **in its own thread**.

Array calls work for tasks just as they do for classes. That is, any **task** method can be called on the array object, and the call will reach each of the cells of the array.

Array calls **do not block** the thread that makes the call. Furthermore, the return object by an array call is the **array** itself, as illustrated by the following line from the example in this section.

```
taskArray.Show().Set().Show();
```

In the above line, three methods in a sequence are called on the array object `taskArray`.

The output, explained after the example, shows two important things. First, the call does not block the main thread. The second point is quite important. When a task object must terminate (goes out of scope, for instance), it **stops accepting new requests**, but **responds to all requests in its queue before terminating**.

The example uses a **dynamic** array, but the output is the same for a static array.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

task SimpleTask

      int i;

public:

      SimpleTask(void);
      ~SimpleTask(void);

      void Show(void);

      void Set(void);
      void Set(int);
end;

// ------------------------------------------------------------------ //

SimpleTask::SimpleTask(void)
      i = 5;
end;
```

```
SimpleTask::~SimpleTask(void)
      output << "Inside SimpleTask::~SimpleTask(void).\n";
end;

void SimpleTask::Show(void)
      output << i << '\n';
end;

void SimpleTask::Set(void)
      i += 17;
end;

void SimpleTask::Set(int k)
      i = k;
end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      int dim = 3;
      SimpleTask taskArray[dim];

// Change initial values of task member

      taskArray[0].Set(7);
      taskArray [1].Set(47);
      taskArray [2].Set(97);

// Array call is none-blocking, and returns array object

      taskArray.Show().Set().Show();

      output << "Good-bye World!\n";

end;
```

The output is as follows.

```
Hello World!
Good-bye World!
97477


114
64
24
Inside SimpleTask::~SimpleTask(void).
Inside SimpleTask::~SimpleTask(void).
Inside SimpleTask::~SimpleTask(void).
```

Explaining the output.

There are four threads in this example, so the output may appear strange.

The main thread ends first. You can see that from the output of "`Good-bye World!`".

The number `97477` is actually three outputs without new-line. First 97, then 47 and finally 7 are output, but the output of new-line for each line is done after each **task** thread gains its time-slice, so you see three blank lines.

The second set of numbers is output as expected. Each **task** thread completes its output before losing its time-slice.

Finally you see the **task** destructors doing their output. It is at that point that **task** objects are destroyed, which is **much later** than the time that the main thread ended. That means, task objects will **respond to all requests already in their queue** before terminating.

Section III.3.F. Task Templates

Task templates are defined the same way **class** templates are defined. In the example of this section we define a **task template**, with invariants, constraints, idler and signal handlers that require **accept** specification. This example uses several advanced abstractions of Z++, in one short program.

In main() we instantiate two instances of the **task template** idlingTask. IntegerIdl is an instantiation using argument **int** for the **template** type parameter, and ShortIdl is an instantiation with type **short**.

The example uses process-bounded entire signals, which must be registered to catch, and all threads registering such signals will eventually catch them.

The output of the program takes time to understand, because the outputs of three threads are inter-mixed. In particular when new line of one thread is printed after the output of another thread.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////
// These signals, once generated can be caught
// by any thread, only once.
// ---------------------------------------------------------------- //

enum MyServiceSignals : signalEventType {
```

```
      _SIGNAL_FirstIdler,
      _SIGNAL_SecondIdler,
      _SIGNAL_ThirdIdler,
      _SIGNAL_FourthIdler
};

////////////////////////////////////////////////////////////////////
// To catch these signals, a thread must register
// for them via accept(...). All threads registered for a signal,
// will catch it once the signal is generated.
// ---------------------------------------------------------------- //

enum MyEntireSignals : threadEntireEventType {
      _THREAD_ENTIRE_FirstIdler,
      _THREAD_ENTIRE_SecondIdler,
      _THREAD_ENTIRE_ThirdIdler
};

////////////////////////////////////////////////////////////////////
//                          The task template
// ---------------------------------------------------------------- //

template<type T> task idlingTask

      T counter; // This task has one data-member only

// When condition of invariant is violated,
// the method trigger() will be invoked.

      invariant (counter == 7) trigger();

// ---------------------------------------------------------------- //
// Methods for use as trigger in case conditions are violated.

      void trigger(void); // invariant trigger
      void trigger(T);    // constraint trigger

// ---------------------------------------------------------------- //
// The idler method automatically runs when task thread
// has nothing to do.

      @idlingTask(void);

// ---------------------------------------------------------------- //
// Accept list of signals from : threadEntireEventType,
// and their handlers.
// ---------------------------------------------------------------- //

      accepts(_THREAD_ENTIRE_FirstIdler, _THREAD_ENTIRE_SecondIdler);

// ---------------------------------------------------------------- //
// Handlers are automatically invoked when their signal arrives.

      void FirstEntireHandler(void)<_THREAD_ENTIRE_FirstIdler>;
      void SecondEntireHandler(void)<_THREAD_ENTIRE_SecondIdler>;

public:
```

```
      idlingTask(T); // constructor

// A method with a single constraint.

      void constrainedMethodInt(T j)
      {
            (j < counter) trigger(j);
      };

end; // end task definition

////////////////////////////////////////////////////////////////////
//                      Definitions for task methods
// ---------------------------------------------------------------- //

template<type T> void idlingTask::trigger(void)
      output << "idlingTask::trigger(void): Invariant violation.\n";
end;

// ---------------------------------------------------------------- //

template<type T> void idlingTask::trigger(T n)
      output << "idlingTask::trigger(n) : n == " << n;
      output << ": Constraint violation.\n";
end;

// ---------------------------------------------------------------- //
// Constructor

template<type T> idlingTask::idlingTask(T k)
      counter = k;
end;

// ---------------------------------------------------------------- //
// Task idler

template<type T> idlingTask::@idlingTask(void)
      output << "Inside idler, with counter = " << counter << '\n';
end;

// ---------------------------------------------------------------- //
// Handlers taking signal from : threadEntireEventType for accepts(...)

template<type T>
void idlingTask::FirstEntireHandler(void)
output << "Inside FirstEntireHandler. Sending _SIGNAL_FirstIdler...\n";
      signal <- _SIGNAL_FirstIdler;
end;

template<type T>
void idlingTask::SecondEntireHandler(void)
output << "Inside SecondEntireHandler. Sending _SIGNAL_SecondIdler.\n";
      signal <- _SIGNAL_SecondIdler;
end;

//----------------------------------------------------------------//
```

```
template<type T>
void idlingTask::constrainedMethodInt(T j)
      output << "Inside constrainedMethodInt(T).\n";
      output << "Argument is : " << j << '\n';
      output << "Counter is : " << counter << '\n';
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Make two instantiations of template task idlingTask.

      output << "Main: creating tasks...\n";

      idlingTask<int> IntegerIdl;   // This invokes invariant trigger
      idlingTask<short> ShortIdl(7);

// Send each of entire signals to the two task objects,
// and wait for response signal from both task objects.

      output << "Main: sending _THREAD_ENTIRE_FirstIdler...\n";
      signal <- _THREAD_ENTIRE_FirstIdler;

      output << "Main: Waiting for 'FIRST' _SIGNAL_FirstIdler...\n";
      do enddo(signal ? _SIGNAL_FirstIdler);
      output << "Main: Waiting for 'SECOND' _SIGNAL_FirstIdler...\n";
      do enddo(signal ? _SIGNAL_FirstIdler);

      output << "Main: sending _THREAD_ENTIRE_SecondIdler...\n";
      signal <- _THREAD_ENTIRE_SecondIdler;

      output << "Main: Waiting for 'FIRST' _SIGNAL_SecondIdler...\n";
      do enddo(signal ? _SIGNAL_SecondIdler);
      output << "Main: Waiting for 'SECOND' _SIGNAL_SecondIdler...\n";
      do enddo(signal ? _SIGNAL_SecondIdler);

// This will violate the constraint, so its trigger will be invoked.

      ShortIdl.constrainedMethodInt(15);

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Main: creating tasks...
idlingTask::trigger(void): Invariant violation.
Inside      idler,      with      counter      =      0Main:      sending
_THREAD_ENTIRE_FirstIdler...
```

```
Main: Waiting for 'FIRST' _SIGNAL_FirstIdler...

Inside FirstEntireHandler. Sending _SIGNAL_FirstIdler...
Main: Waiting for 'SECOND' _SIGNAL_FirstIdler...
Inside FirstEntireHandler. Sending _SIGNAL_FirstIdler...
Inside idler, with counter = Inside idler, with counter = 7Main:
sending _THREAD_ENTIRE_SecondIdler...
0

Inside idler, with counter = Main: Waiting for 'FIRST'
_SIGNAL_SecondIdler...
Inside SecondEntireHandler. Sending _SIGNAL_SecondIdler.
7
Main: Waiting for 'SECOND' _SIGNAL_SecondIdler...
Inside idler, with counter = 0Inside SecondEntireHandler. Sending
_SIGNAL_SecondIdler.

Inside idler, with counter = 0idlingTask::trigger(n) : n == 15:
Constraint violation.

Inside constrainedMethodInt(T).
Argument is : Inside idler, with counter = 0
15
Inside idler, with counter = Counter is : 70

Inside idler, with counter = Inside idler, with counter = 7Good-bye
World!
0
```

Section III.3.G. Exceptions in Tasks

An exception raised (and not repaired) in a **task** thread is **reported** to the parent thread. All usual **exception clean up** is done, but the **task** object (and its thread) are **not** destroyed. This is illustrated by the following example.

```
// Example.zpp

#include<iostream.h>
#include<exception.h>

using namespace ioSpace;
using namespace exceptionSpace;

// ------------------------------------------------------------------ //
// Extend system exceptions

enum userExceptionEvents : exceptionEventType {
                                _EXCEPTION_FirstUserException,

_EXCEPTION_SecondUserException
};

// ------------------------------------------------------------------ //

task exceptionThreadType
      int counter;
public:
      exceptionThreadType(void);
      void UserEventGenerator(int);
end;

// ------------------------------------------------------------------ //

exceptionThreadType::exceptionThreadType(void)
      counter = 0;
end;

// ------------------------------------------------------------------ //

void exceptionThreadType::UserEventGenerator(int i)
            throws(_EXCEPTION_SecondUserException)

      output << "Entering UserEventGenerator()\n";

// Next line may raise exception

      if (i < 47) raise _EXCEPTION_SecondUserException; endif;

// Next line will not print if exception is raised

      output << "Leaving UserEventGenerator()\n";
end;
```

```
////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

  output << "Hello World!\n";

  exceptionThreadType ETT;

  layer<userExceptionEvents>

// Next line will raise exception in thread of task object ETT.

    ETT.UserEventGenerator(7);

// Next line will not execute

    output << "Layer completed.\n";

  handler

  case _EXCEPTION_FirstUserException .. _EXCEPTION_SecondUserException:
    output << "Caught User-Defined Exception.\n";

  endlayer;

  output << "Outside of the exception layer.\n";

// Task object ETT is not destroyed for raising exception

  ETT.UserEventGenerator(97);

  output << "Good-bye World!\n";

end;
```

The output of the example is as follows.

```
Hello World!
Entering UserEventGenerator()
Caught User-Defined Exception.
Outside of the exception layer.
Entering UserEventGenerator()
Leaving UserEventGenerator()
Good-bye World!
```

Chapter III.4. Components and Modules

A component is a piece of software, with a well-defined boundary, that can be used in building larger software, solely through its boundary. The process of composing larger software from smaller ones is inherently recursive. That is, the larger software, usually called an application, is technically a component. However, it may not have been designed for use as a component in composing larger applications.

A component must have a well-defined boundary in the form of a so-called contract. This allows modifying or replacing a component without affecting the larger programs that use it. In other words, **a component hides its internals within its boundary**.

The boundary of a Z++ program is the set of its **entry** points. **Any Z++ program is a component**. Users of a component can only communicate with it through its boundary, i.e. its **entry** points, making the internals of the component completely hidden away from its users.

In a distributed setting, components of an application may reside on different nodes in a network. That is, there is no requirement for a component to be attached to the larger application that uses it. This observation is the distinction between software components and hardware components.

The type constructor **module** is the mechanism for (distributed) component-oriented development. While any Z++ program is a component, we need a linguistic abstraction in order to compose a larger Z++ program from existing smaller ones.

The type constructor module uses **class** or **task** type constructors to **represent** a component in a Z++ program. This will be illustrated in the following subsections. The only feature not allowed for modules is the notion of **template**. Modules do not lend themselves to clear semantics with regard to templates. In particular, a component is a complete program, not a type.

Section III.4.A. Local Modules

The terms **local** and **remote** is about how we use components in an application, and **not** the form or **implementation** of components. This will become clear in later sections.

Modular examples naturally involve two Z++ programs. One program will act as a component, and the other program will use the notion of **module** to represent this component. For clarity, we shall refer to these programs as Component and Module, respectively. That is, Module is the Z++ program that will use the Z++ program called Component via the notion of module.

Below is an example for use as a component in a module listed following it.

```
// Component.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

struct object
      int i;
      double d;

      object(int, double);
end;

object::object(int n, double b)
      i = n;
      d = b;
end;

////////////////////////////////////////////////////////////////////
// When this program is executed on its own, the main() entry point
// is entered.
//-------------------------------------------------------------------//
// When used as a component, all entry points can be entered by
// the program loading this component.
////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "I must be loaded as a component!\n";
end;

entry int FirstEntry(int n)
      output << "FirstEntry : " << n << '\n';
      return 2 * n;
end;

entry double SecondEntry(double d)
      output << "SecondEntry : " << d << '\n';
      return d / 2;
end;
```

```
entry short ThirdEntry(short s)
      output << "ThirdEntry : " << s << '\n';
      return s + 11;
end;

entry char FourthEntry(char c)
      output << "FourthEntry : " << c << '\n';
      return ++c;
end;

entry object FifthEntry(object b)
      output << "FifthEntry : got object with values " <<
           b.i << " and " << b.d << '\n';

      output << "FifthEntry : creating new object for return.\n";
      object t(7, 47.97);
      return t;
end;
```

Now a program using the above program as a **component**. This example illustrates the following points.

A **class** or **task** becomes a module when the name of the **class/task** is followed by the operator = (assignment) and a **literal** string. The string is the path to the executable of the component. Note that we called the previous example Component.zpp, and we are assuming its executable is named Component.zxe. The symbols preceding the name of the executable are intended to mean "the path to reach the executable", and are not part of the syntax.

A module, just like a **class** or **task**, can have members and methods. Among the methods of a module those specified **external** are the gateway to the boundary of the component being represented by the module. Note that **external** and **extern** are two distinct notions.

Compiler generates the body of an external method. You only provide a prototype.

Notice that methods of the module named `LocalModule` are precisely the **entry** points of the component represented by this module. A module is free to use any number of the **entry** points that constitute the boundary of a component.

Compiler cannot check the validity of signature for an **external** method of a module. In practice, the implementer of a component can provide a header file including the definition of a generic module with nothing but the **external** methods to interact with the component. Users of the module can use the definition directly, or derive from this generic module and adjust its use as they need. For instance, one can add invariants and constraints for calling the methods, as will be illustrated later.

```
// Module.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

struct object
      int i;
      double d;

      object(int, double);
end;

object::object(int n, double b)
      i = n;
      d = b;
end;

//-------------------------------------------------------------------//
// The string "~/Component.zxe" is full-pathname of the program
// we called Component.zxe before this program.
// LocalModule is an example of notion of module in Z++. Its name
// is followed with assignment operator, and a literal string. It
// also has methods specified as external.
//-------------------------------------------------------------------//

struct LocalModule = "~/Component.zxe"
      int i;

// Compiler defines the bodies of external methods

      external int FirstEntry(int);
      external double SecondEntry(double);
      external short ThirdEntry(short);
      external char FourthEntry(char);
      external object FifthEntry(object);

      LocalModule(void);

end;

LocalModule::LocalModule(void)
      i = 3;
end;

////////////////////////////////////////////////////////////////////
//-------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      LocalModule LM; // Create an instance of module
```

```
// Now call the methods of LM specified as external. Each such
// call invokes a corresponding entry point of Component.zxe.

      output << "Main: FirstEntry returned: " <<
          LM.FirstEntry(20) << '\n';

      double dblRet = LM.SecondEntry(21.12);
      output << "Main: SecondEntry returned: " << dblRet << '\n';

      short shortRet = LM.ThirdEntry(5);
      output << "Main: ThirdEntry returned: " << shortRet << '\n';

      char charRet = LM.FourthEntry('X');
      output << "Main: FourthEntry returned: " << charRet << '\n';

      output << "Main: creating object for sending to FifthEntry.\n";
      object t(3, 8.5);

      t = LM.FifthEntry(t);

      output << "Main: FifthEntry returned object with values " <<
          t.i << " and " << t.d << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows. Note that the output is a mixture of outputs of the two programs.

```
Hello World!
Main: FirstEntry returned: FirstEntry : 20
40
SecondEntry : 21.12
Main: SecondEntry returned: 10.56
ThirdEntry : 5
Main: ThirdEntry returned: 16
FourthEntry : X
Main: FourthEntry returned: Y
Main: creating object for sending to FifthEntry.
FifthEntry : got object with values 3 and 8.5
FifthEntry : creating new object for return.
Main: FifthEntry returned object with values 7 and 47.97
Good-bye World!
```

Section III.4.B. Loading and unloading Modules

The executable component represented by a module is loaded at time of declaration of an instance of the module. Once a module is loaded, any further instances created will **share** the same executable, but will have **their own state and context**.

When an instance of a module goes out of scope its state is destroyed. The executable component is unloaded when the last instance of the module representing it goes out of scope, or is destroyed in case of dynamic instances.

In the example of previous section, the component is loaded and initialized at the following declaration statement. Much of initialization involves the creation of global objects.

```
LocalModule LM;
```

It is unloaded after main(), function in which it was declared, ends. That is when the instance LM goes out of scope and is destroyed.

Specifically, there are no instructions in Z++ language for loading and unloading modules. Also, as mentioned earlier, Z++ compiler generates the body of **external** methods. This is true whether modules are loaded locally or remotely. This design avoids a number of errors when the engineer is made responsible for writing none-trivial code and bookkeeping.

Section III.4.C. Module Context

For each instance of a module, at elaboration (declaration), the global state of the component (represented by the instance) is created. The component remains dormant until an **external** method of the instance interacts with it. After returning from the **entry** point invoked by the **external** method, module becomes dormant retaining the global state that it has reached, until the next call to one of its **entry** points.

The example in previous section provides the simplest view for describing **module context switch**. There is one single thread of execution. In this simple case, at invocation of an **external** method, the thread of execution leaves the context of the program called Module (in previous example) and enters the context of the program called Component. At this point we usually say that the module has been invoked, though it is really the component represented by the module that has been awakened.

The context switch at module invocation has the same meaning as thread context switch, though with some differences. Even in a single-threaded program, as in the previous example, a form of context switch does take place, when the thread of control moves from one program (Module) to another (Component) and back. We refer to this as **module context switch**.

Module context switch is independent of thread context switch. The latter is associated with time slice for executing instructions of Z47 Processor. In a multi-threaded Module program, the thread that invokes a module will have a module context switch as it enters the code of the Component, and when it returns to Module. This module context switch is not affected by time-slice associated with thread context switch. The invoking thread can be in its own context (the Module) or the context of the invoked component (the Component) when it receives its time slice.

Section III.4.D. Types of arguments and return object

As we shall see, switching between local and remote loading of a module takes a small code change at the definition of a module. This simple localized change is the whole purpose of the design of Z++. The requirement for maintaining this simplicity is to disallow pass by reference and passing pointers to **entry** points even in case of local loads. This limitation avoids any code change to modules when switching between local and remote loads.

All built-in types and **class** types that do not contain pointers can be used for types of parameters of an **entry** point, and its return object. In the example of this section, we pass and receive objects of type **string**, **enum** and a **class** with such members. In addition, regular (static) arrays of these types are allowed for arguments.

Type constructor **collection** uses hidden pointers. Besides, instances of a **collection** are too large and complex for passing among modules, as are instances of type constructor **task**. Note that the problem is the semantics of such operations, otherwise Z++ processes move among nodes retaining their state and context (Autonomous Agents). Passing a pointer to an object in the context of one Z47 to another Z47 has no meaning.

The following example consists of a Component, and then a Module that loads the executable of this Component.

```
// Component.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

enum componentEnum {_componentOne, _componentTwo, _componentThree};

//-------------------------------------------------------------------//

struct stringStruct
      string stg;
      componentEnum ce;

      stringStruct(string, componentEnum);
end;

stringStruct::stringStruct(string s, componentEnum e)
      stg = s;
      ce = e;
end;

////////////////////////////////////////////////////////////////////
//-------------------------------------------------------------------//

entry void main(void)
```

```
      output << "Hello World!\n";
      output << "Good-bye World!\n";
end;

entry string FirstEntry(componentEnum e)
      output << "FirstEntry: got enum: " << [e] << '\n';
      return "String from FirstEntry.";
end;

entry componentEnum SecondEntry(string s)
      output << "SecondEntry: got string: " << s << '\n';
      return _componentTwo;
end;

entry stringStruct ThirdEntry(stringStruct ss)
      output << "ThirdEntry: got stringStruct. String is: ";
      output << ss.stg << " and enum is: " << [ss.ce] << '\n';
      output << "Creating new stringStruct for return.\n";
      stringStruct ssr("ThirdEntry", _componentThree);
      return ssr;
end;
```

This is the Module example for loading (the executable of) the above Component.

```
// Module.zpp

#include<iostream.h>
using namespace ioSpace;

//---------------------------------------------------------------------//

enum componentEnum {_componentOne, _componentTwo, _componentThree};

//---------------------------------------------------------------------//

struct stringStruct
      string stg;
      componentEnum ce;

      stringStruct(string, componentEnum);
end;

stringStruct::stringStruct(string s, componentEnum e)
      stg = s;
      ce = e;
end;

/////////////////////////////////////////////////////////////////////
// The path to Component.zxe must be its actual path.

struct LocalModule = "~/Component.zxe"

      external string FirstEntry(componentEnum);
      external componentEnum SecondEntry(string);
      external stringStruct ThirdEntry(stringStruct);
```

```
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      LocalModule LM;

      string stg = LM.FirstEntry(_componentOne);
      output << "Main: got string " << stg << '\n';

      componentEnum cen = LM.SecondEntry("Coming from Main.");
      output << "Main: got enum " << [cen] << '\n';

      stringStruct sst("Hello from Main.", _componentOne);
      sst = LM.ThirdEntry(sst);
      output << "Main: " << sst.stg << '\n';
      output << "Main: " << [sst.ce] << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
FirstEntry: got enum: 0
Main: got string String from FirstEntry.
SecondEntry: got string: Coming from Main.
Main: got enum 1
ThirdEntry: got stringStruct. String is: Hello from Main. and enum is:
0
Creating new stringStruct for return.
Main: ThirdEntry
Main: 2
Good-bye World!
```

Section III.4.E. Module and Invariants

An **external** method executes in the context of a different program and has no effect on the state of the module owning it. However, the return object of an **external** method may be used in various ways to change the state of its module. Besides, a module may need **invariant**s for other reasons.

In the presence of **invariant**s **external** methods cannot be **public**. Simply make **public** methods that call the **external** methods, as shown in the following example.

The example, like all examples in this section consists of two programs. Following the same pattern, we call the first program Component, and the second program that loads it, the Module.

```
// Component.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";
      output << "Good-bye World!\n";
end;

entry int IntEntry(int n)
      return n + 2;
end;

entry double DoubleEntry(double d)
      return d * 2;
end;
```

Here is the Module program that loads the executable of the above Component. As shown, the **external** method `IntEntry(int)` violates **invariants** and **triggers** the method `IntTrigger()`, which corrects the value of the member i.

```
// Module.zpp

#include<iostream.h>
using namespace ioSpace;

//--------------------------------------------------------------------//
// For actual run the complete path to component is needed.

class LocalModule = "~/Component.zxe"

      invariant(i < 7) IntTrigger();
```

```
      invariant(d < 100) DoubleTrigger();

      int i;
      double d;

// When class has invariants, external methods cannot be public

      external int IntEntry(int);
      external double DoubleEntry(double);

      void IntTrigger(void);
      void DoubleTrigger(void);

public:

      LocalModule(void);

// These public methods call external methods, and invariants
// are tested for these public methods.

      void IntEntryCaller(int);
      double DoubleEntryCaller(double);

end;

LocalModule::LocalModule(void)
      i = 77;
      d = 5.2;
end;

void LocalModule::IntEntryCaller(int n)
      i = IntEntry(n); // call external method
end;

double LocalModule::DoubleEntryCaller(double b)
      return d = DoubleEntry(b); // call external method
end;

void LocalModule::IntTrigger(void)
  output << "LocalModule::IntTrigger(): correcting i member value.\n";
  i = 77;
end;

void LocalModule::DoubleTrigger(void)
output << "LocalModule::DoubleTrigger(): correcting d member value.\n";
  d = 5.2;
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      LocalModule LM;
```

```
        LM.IntEntryCaller(3);
        LM.DoubleEntryCaller(33.57);

        output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
LocalModule::IntTrigger(): correcting i member value.
Good-bye World!
```

Section III.4.F. External methods and Constraints

At times we may need to constrain the input to an **external** method. To do so, use the constraints on a regular method that simply calls the **external** method.

In the following example we are using the same Component as in the previous section on **invariants**. The Module program illustrates the use of constraints for **external** methods.

In calling `DoubleEntryCaller()` the constraint is violated, and the trigger is invoked. Note that in this simple case, `DoubleTrigger()` only prints a message and returns to the method that invoked it. Thus, the call to the **external** method will be made with an invalid argument.

```
// Module.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

struct LocalModule = "~/Component.zxe"

      external int IntEntry(int);
      external double DoubleEntry(double);

      int IntEntryCaller(int n)
      {
            (n > 5) IntTrigger(n);
      };

      double DoubleEntryCaller(double b)
      {
            (b < 50) DoubleTrigger(b);
      };

private:
      void IntTrigger(int);
      void DoubleTrigger(double);
end;

// This is a constrained method that calls an external method

int LocalModule::IntEntryCaller(int n)
      return IntEntry(n); // call external method
end;

// This is a constrained method that calls an external method

double LocalModule::DoubleEntryCaller(double b)
      return DoubleEntry(b); // call external method
end;

void LocalModule::IntTrigger(int n)
```

```
      output << "IntTrigger(). Argument too small: " << n << '\n';
end;

void LocalModule::DoubleTrigger(double b)
      output << "DoubleTrigger(). Argument too large: " << b << '\n';
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      LocalModule LM;

      LM.IntEntryCaller(10);
      LM.DoubleEntryCaller(67.89);

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
DoubleTrigger(). Argument too large: 67.89
Good-bye World!
```

Section III.4.G. Module Inheritance

Module is either a **class** or a **task**, and therefore inheritance for modules follows the same rules. For the sake of illustration, here we make four programs. The first three are components that the fourth program, Module.zpp loads them as modules. Further explanations are given at Module.zpp.

Here is the first component.

```
// ComponentOne.zpp

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
end;

entry int EntryInt(int n)
      return ++n;
end;

entry short EntryShort(short s)
      return s + 3;
end;
```

Next program is the second component.

```
// ComponentTwo.zpp

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
end;

entry double EntryDouble(double d)
      return d * 2;
end;

entry float EntryFloat(float f)
      return f + 1.1;
end;
```

Below is the third component.

```
// ComponentThree.zpp

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
end;

entry char EntryChar(char c)
      return ++c;
end;
```

This is the program that uses the above three components.

In the program, the two modules `ModuleOne` and `ModuleTwo` represent components One and Two above. Then `DerivedModule` represents the Third component and derives from the other two modules.

Note that for real run you will need to provide the full path to executables.

In the main() **entry** point we create an instance of `DerivedModule` DM. At this point, as the instance DM is being constructed all three components are loaded. The components are unloaded after the main() **entry** points ends. That is, when the instance DM goes out of scope.

The inherited **external** methods can be called just like regular methods. You can use the member notation, or reach the base and then call the method, as illustrated.

```
// Module.zpp

#include<iostream.h>
using namespace ioSpace;

//--------------------------------------------------------------------//
// ModuleOne represents ComponentOne.zxe.
//--------------------------------------------------------------------//

struct ModuleOne = "~/ComponentOne.zxe"

      external int EntryInt(int);
      external short EntryShort(short);
end;

//--------------------------------------------------------------------//
// ModuleTwo represents ComponentTwo.zxe.
//--------------------------------------------------------------------//

struct ModuleTwo = "~/ComponentTwo.zxe"
```

```
      external double EntryDouble(double);
      external float EntryFloat(float);
end;

//-------------------------------------------------------------------//
// DerivedModule represents ComponentThree.zxe, but it also derives
// from ModuleOne and ModuleTwo. Thus, DerivedModule represents
// all three components.
//-------------------------------------------------------------------//

struct DerivedModule = "~/ComponentThree.zxe"
      : ModuleOne, ModuleTwo

      external char EntryChar(char);
end;

/////////////////////////////////////////////////////////////////////
//-----------------------------------------------------------------//
/////////////////////////////////////////////////////////////////////

entry void main(void)
  output << "Hello World!\n";

  DerivedModule DM; // All three components are loaded here

  output << "EntryInt() returned: " << DM.EntryInt(76) << '\n';

  output << "EntryShort() returned: " <<
      DM::ModuleOne::EntryShort(7) << '\n';

  output << "EntryDouble() returned: " << DM.EntryDouble(8.2) << '\n';

  output << "EntryFloat() returned: " <<
      DM::ModuleTwo::EntryFloat(7.3) << '\n';

  output << "EntryChar() returned: " << DM.EntryChar('Y') << '\n';

  output << "Good-Bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
EntryInt() returned: 77
EntryShort() returned: 10
EntryDouble() returned: 16.4
EntryFloat() returned: 8.4
EntryChar() returned: Z
Good-Bye World!
```

Section III.4.H. Remote Linkage

Suppose we are dealing with two nodes (computers). Call one of them Local-Node and the other Remote-Node. Assume program Calling-Program resides on Local-Node, and program Component resides on Remote-Node. Finally, suppose we wish that the Calling-Program during its execution to load and use the program Component. In this scenario, Component is a **remote component**, and Calling-Program uses **remote linkage** to load and use Component.

Remote linkage takes two distinct forms. The Calling-Program can download the Component to Local-Node and execute it there. However, the Calling-Program can also have the program Component execute on Remote-Node, and use it just the same.

In Z++ the terms **local** and **remote** as used for loading components are relative to the **location** of loading a **remote component**. That is, specifying a **local** load means that we wish to **download** the component to the **local node** for execution, while specifying **remote** means that we wish the component to **execute** on the **remote node** where it resides.

In previous sections the loading and use of local components was illustrated. In those examples, both the calling program and the component resided on the same node. The same pattern and syntax holds for remote linkage. That is, the notion of module is used via the **class** or **task** type constructors. The only difference is in the literal **string** coming after the assignment symbol =. For remote linkage, we need to add a prefix to the full pathname to reach the file. This prefix is the **red string** in the following example.

```
"Zpp://IP-Address/full-path/ComponentName.zxe"
```

The IP-Address is a string in the form of `"xxx.xxx.xxx.xxx"` and is the address to reach the **Z++ Internet Server** running on the remote node.

The next two sections illustrate the semantics of the terms **local** and **remote**. The **default** load of a remote module is **local**.

Section III.4.H.1. Local loading of a Remote Component

There are three files in this example, as follows.

RemoteComponent.h is a header included in the other two files. It defines the types of arguments and return objects to the **entry** points.

RemoteCalledModule.zpp is the program that its executable, RemoteComponent.zxe resides on a remote node. This is loaded as a component.

RemoteCallingModule.zpp is the program that runs on the local machine and loads the program RemoteComponent.zxe as a component.

```
//RemoteComponent.h

struct leftBaseType
      long g;

      leftBaseType(long);
end;

leftBaseType::leftBaseType(long l)
       g = l;
end;

////////////////////////////////////////////////////////////////////////

struct rightBaseType
      short t;

      rightBaseType(short);
end;

rightBaseType::rightBaseType(short s)
       t = s;
end;

////////////////////////////////////////////////////////////////////////

struct member
      double d;

      member(double);
end;

member::member(double b)
      d = b;
end;

////////////////////////////////////////////////////////////////////////

struct bigArgument : leftBaseType, rightBaseType
      int n;
```

```
      member M;

      bigArgument(int, long, short, double);
end;

bigArgument::bigArgument(int b, long l, short r, double m) :
            leftBaseType(l), rightBaseType(r), M(m)
      n = b;
end;

////////////////////////////////////////////////////////////////////////

struct returnObject
      long v;

      returnObject(void);
end;

returnObject::returnObject(void)
      v = 1313;
end;

////////////////////////////////////////////////////////////////////////

struct stgArgument
      string st;

      stgArgument(void);
end;

stgArgument::stgArgument(void)
      st = "I am default string.";
end;

////////////////////////////////////////////////////////////////////////

struct multiStringArgument
      string first;
      string second;
      double dbl;
      string last;

      multiStringArgument(void);
end;

multiStringArgument::multiStringArgument(void)
      first = "I am first string.";
      second = "I am second string.";
      dbl = 456.123;
      last = "I am last one.";
end;

////////////////////////////////////////////////////////////////////////

struct memberStringArgument
      stgArgument sa;
      string st;
```

```
      multiStringArgument ma;

      memberStringArgument(void);
end;

memberStringArgument::memberStringArgument(void)
      st = "I am the self string member.";
end;

////////////////////////////////////////////////////////////////////////

struct derivedStringArgument : memberStringArgument
end;

////////////////////////////////////////////////////////////////////////

struct baseMemberStringArgument :
            derivedStringArgument, memberStringArgument
      memberStringArgument msa;
      multiStringArgument sma;
      derivedStringArgument dsa;
end;

////////////////////////////////////////////////////////////////////////

struct arrayStringArgument
      memberStringArgument asa[3];

end;

////////////////////////////////////////////////////////////////////////

struct arrayStringMember
      string asm[4][2];

      arrayStringMember(void);
end;

arrayStringMember::arrayStringMember(void)
      asm[0][0] = "Joe";
      asm[0][1] = "Joe Smith";
      asm[1][0] = "Jim";
      asm[1][1] = "Jim Black";
      asm[2][0] = "Jack";
      asm[2][1] = "Jack White";
      asm[3][0] = "John";
      asm[3][1] = "John Brown";
end;

////////////////////////////////////////////////////////////////////////

enum remoteEnumes {_apple, _cherry, _orange};

////////////////////////////////////////////////////////////////////////

struct enumClassArgType
      int t;
```

```
      remoteEnumes e;

      enumClassArgType(void);
end;

enumClassArgType::enumClassArgType(void)
      t = 67;
      e = _cherry;
end;

////////////////////////////////////////////////////////////////////////
```

The next program is loaded as a component. It consists of a large number of **entry** points that are called by programs that load it.

We assume the executable of this program is called RemoteComponent.zxe.

```
//RemoteCalledModule.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// Use correct path to reach the include file.

#include"~/RemoteComponent.h"

// ------------------------------------------------------------------ //

entry void main(void)
      output << "Main: Hello World!\n";
end;

// ------------------------------------------------------------------ //

entry void First(void)
      output << "First: Hello World!\n";
end;

// ------------------------------------------------------------------ //

entry void Second(void)
      output << "Second: Hello World!\n";
end;

// ------------------------------------------------------------------ //

entry void Third(int i)
      output << "Third: the argument is " << i << '\n';
end;

// ------------------------------------------------------------------ //
```

```
entry void Fourth(bigArgument b)
      output << "Starting Fourth ====>\n\n";
      output << "bigArgument.n : " << b.n << '\n';
      output << "bigArgument.M.d : " << b.M.d << '\n';
      output << "bigArgument::leftBaseType::g : " <<
            b::leftBaseType::g << '\n';

      output << "bigArgument::rightBaseType::t : " <<
            b::rightBaseType::t << '\n';

      output << "\nReturning from Fourth <====\n";
end;

// ------------------------------------------------------------------ //

entry void Fifth(bigArgument a, bigArgument b)
      output << "Starting Fifth ====>\n\n";

      output << "Printing First argument ====>\n\n";

      output << "bigArgument.n : " << a.n << '\n';
      output << "bigArgument.M.d : " << a.M.d << '\n';
      output << "bigArgument::leftBaseType::g : " <<
            a::leftBaseType::g << '\n';

      output << "bigArgument::rightBaseType::t : " <<
            a::rightBaseType::t << '\n';

      output << "Printing Second argument ====>\n\n";

      output << "bigArgument.n : " << b.n << '\n';
      output << "bigArgument.M.d : " << b.M.d << '\n';
      output << "bigArgument::leftBaseType::g : " <<
            b::leftBaseType::g << '\n';

      output << "bigArgument::rightBaseType::t : " <<
            b::rightBaseType::t << '\n';

      output << "\nReturning from Fifth <====\n";
end;

// ------------------------------------------------------------------ //

entry int Sixth(void)
      output << "Starting Sixth ====>\n\n";
      output << "\nReturning from Sixth <====\n";
      return 57;
end;

// ------------------------------------------------------------------ //

entry double Seventh(bigArgument a)
      output << "Starting Seventh ====>\n\n";
      output << "Printing argument ====>\n\n";

      output << "bigArgument.n : " << a.n << '\n';
      output << "bigArgument.M.d : " << a.M.d << '\n';
```

```
      output << "bigArgument::leftBaseType::g : " <<
            a::leftBaseType::g << '\n';

      output << "bigArgument::rightBaseType::t : " <<
            a::rightBaseType::t << '\n';

      output << "\nReturning from Seventh <====\n";
      return 75.93;
end;

// ------------------------------------------------------------------ //

entry returnObject Eighth(bigArgument b)
      output << "Starting Eighth ====>\n\n";
      output << "Printing argument ====>\n\n";

      output << "bigArgument.n : " << b.n << '\n';
      output << "bigArgument.M.d : " << b.M.d << '\n';
      output << "bigArgument::leftBaseType::g : " <<
                              b::leftBaseType::g << '\n';

      output << "bigArgument::rightBaseType::t : " <<
                              b::rightBaseType::t << '\n';

      returnObject rto;
      rto.v = 747;

      output << "\nReturning from Eighth <====\n";

      return rto;
end;

// ------------------------------------------------------------------ //

entry void Nineth(stgArgument s)
      output << "Starting Nineth ====>\n\n";
      output << "Printing argument ====>\n\n";

      output << s.st << '\n';
      output << "\nReturning from Nineth <====\n";
end;

// ------------------------------------------------------------------ //

entry void Tenth(stgArgument s, multiStringArgument t)
      output << "Starting Tenth ====>\n\n";
      output << "Printing argument ====>\n\n";

      output << s.st << '\n';

      output << t.first << '\n';
      output << t.second << '\n';
      output << t.dbl << '\n';
      output << t.last << '\n';

      output << "\nReturning from Tenth <====\n";
end;
```

```
// ------------------------------------------------------------- //

entry void Eleventh(multiStringArgument t,
                    stgArgument s, memberStringArgument m)
      output << "Starting Eleventh ====>\n\n";

      output << "Printing multiStringArgument ====>\n\n";

      output << t.first << '\n';
      output << t.second << '\n';
      output << t.dbl << '\n';
      output << t.last << '\n';

      output << "Printing stgArgument ====>\n\n";

      output << s.st << '\n';

      output << "Printing memberStringArgument ====>\n\n";

      output << m.sa.st << '\n';

      output << m.st << '\n';

      output << m.ma.first << '\n';
      output << m.ma.second << '\n';
      output << m.ma.dbl << '\n';
      output << m.ma.last << '\n';

      output << "\nReturning from Eleventh <====\n";
end;

// ------------------------------------------------------------- //

entry void Twelvth(derivedStringArgument d)
      output << "Starting Twelvth ====>\n\n";

      output << d::memberStringArgument::sa.st << '\n';

      output << d::memberStringArgument::st << '\n';

      output << d::memberStringArgument::ma.first << '\n';
      output << d::memberStringArgument::ma.second << '\n';
      output << d::memberStringArgument::ma.dbl << '\n';
      output << d::memberStringArgument::ma.last << '\n';

      output << "\nReturning from Twelvth <====\n";
end;

// ------------------------------------------------------------- //

entry void Thirteen(baseMemberStringArgument bm)
      output << "Starting Thirteen ====>\n\n";

      output << "Priting base derivedStringArgument ====>\n\n";

      output <<
```

```
      bm::derivedStringArgument::memberStringArgument::sa.st <<
      '\n';

output <<
      bm::derivedStringArgument::memberStringArgument::st <<
      '\n';

output <<
      bm::derivedStringArgument::memberStringArgument::ma.first
      <<'\n';

output <<
      bm::derivedStringArgument::memberStringArgument::ma.second
      << '\n';

output <<
      bm::derivedStringArgument::memberStringArgument::ma.dbl
      << '\n';

output <<
      bm::derivedStringArgument::memberStringArgument::ma.last
      << '\n';

output << "Priting base memberStringArgument ====>\n\n";

output << bm::memberStringArgument::sa.st << '\n';
output << bm::memberStringArgument::st << '\n';
output << bm::memberStringArgument::ma.first << '\n';
output << bm::memberStringArgument::ma.second << '\n';
output << bm::memberStringArgument::ma.dbl << '\n';
output << bm::memberStringArgument::ma.last << '\n';

output << "Priting member memberStringArgument ====>\n\n";

output << bm.msa.sa.st << '\n';
output << bm.msa.st << '\n';
output << bm.msa.ma.first << '\n';
output << bm.msa.ma.second << '\n';
output << bm.msa.ma.dbl << '\n';
output << bm.msa.ma.last << '\n';

output << "Priting member multiStringArgument ====>\n\n";

output << bm.sma.first << '\n';
output << bm.sma.second << '\n';
output << bm.sma.dbl << '\n';
output << bm.sma.last << '\n';

output << "Priting member derivedStringArgument ====>\n\n";

output << bm.dsa::memberStringArgument::sa.st << '\n';
output << bm.dsa::memberStringArgument::st << '\n';
output << bm.dsa::memberStringArgument::ma.first << '\n';
output << bm.dsa::memberStringArgument::ma.second << '\n';
output << bm.dsa::memberStringArgument::ma.dbl << '\n';
output << bm.dsa::memberStringArgument::ma.last << '\n';
```

```
      output << "\nReturning from Thirteen <====\n";
end;

// ------------------------------------------------------------------ //

entry void Fourteen(baseMemberStringArgument bm,
                            derivedStringArgument d,
                            memberStringArgument m)
end;

// ------------------------------------------------------------------ //

entry int Fifteen(string a, baseMemberStringArgument bm,
            string c, string d)

      output << "Starting Fifteen ====>\n\n";

      output << "First the big argument ====>\n\n";

      output << "Priting base derivedStringArgument ====>\n\n";

      output <<
            bm::derivedStringArgument::memberStringArgument::sa.st
            << '\n';

      output <<
            bm::derivedStringArgument::memberStringArgument::st
            << '\n';

      output <<
            bm::derivedStringArgument::memberStringArgument::ma.first
            << '\n';

      output <<
            bm::derivedStringArgument::memberStringArgument::ma.second
            << '\n';

      output <<
            bm::derivedStringArgument::memberStringArgument::ma.dbl
            << '\n';

      output <<
            bm::derivedStringArgument::memberStringArgument::ma.last
            << '\n';

      output << "Priting base memberStringArgument ====>\n\n";

      output << bm::memberStringArgument::sa.st << '\n';
      output << bm::memberStringArgument::st << '\n';
      output << bm::memberStringArgument::ma.first << '\n';
      output << bm::memberStringArgument::ma.second << '\n';
      output << bm::memberStringArgument::ma.dbl << '\n';
      output << bm::memberStringArgument::ma.last << '\n';

      output << "Priting member memberStringArgument ====>\n\n";

      output << bm.msa.sa.st << '\n';
```

```
	output << bm.msa.st << '\n';
	output << bm.msa.ma.first << '\n';
	output << bm.msa.ma.second << '\n';
	output << bm.msa.ma.dbl << '\n';
	output << bm.msa.ma.last << '\n';

	output << "Priting member multiStringArgument ====>\n\n";

	output << bm.sma.first << '\n';
	output << bm.sma.second << '\n';
	output << bm.sma.dbl << '\n';
	output << bm.sma.last << '\n';

	output << "Priting member derivedStringArgument ====>\n\n";

	output << bm.dsa::memberStringArgument::sa.st << '\n';
	output << bm.dsa::memberStringArgument::st << '\n';
	output << bm.dsa::memberStringArgument::ma.first << '\n';
	output << bm.dsa::memberStringArgument::ma.second << '\n';
	output << bm.dsa::memberStringArgument::ma.dbl << '\n';
	output << bm.dsa::memberStringArgument::ma.last << '\n';

	output << "Now three string arguments ====>\n\n";

	output << a << '\n';
	output << c << '\n';
	output << d << '\n';

	output << "\nReturning from Fifteen <====\n";

	return 56;
end;

// ------------------------------------------------------------------ //

entry void Sixteen(arrayStringArgument a)
	output << "Starting Sixteen ====>\n\n";

	for (int i = 0; i < 3; i++)
		output << "Index is : " << i << '\n';
		output << a.asa[i].sa.st << '\n';
		output << a.asa[i].st << '\n';
		output << a.asa[i].ma.first << '\n';
		output << a.asa[i].ma.second << '\n';
		output << a.asa[i].ma.dbl << '\n';
		output << a.asa[i].ma.last << '\n';
	endfor;

	output << "\nReturning from Sixteen <====\n";
end;

// ------------------------------------------------------------------ //

entry void Seventeen(arrayStringMember a)
	output << "Starting Seventeen ====>\n\n";

	for (int i = 0; i < 4; i++)
```

```
        for (int j = 0; j < 2; j++)
         output << "Index i is : " << i << " and j is : " << j << '\n';
         output << a.asm[i][j] << '\n';
        endfor;
      endfor;

      output << "\nReturning from Seventeen <====\n";
end;

// ------------------------------------------------------------------ //

entry stgArgument Twenty(void)
      output << "Starting Twenty ====>\n\n";

      stgArgument stg;
      stg.st = "I have been changed by remote server.";

      output << "\nReturning from Twenty <====\n";

      return stg;
end;

// ------------------------------------------------------------------ //

multiStringArgument GlobalMulStgArg;
memberStringArgument GlobalMemStgArg;

// ------------------------------------------------------------------ //

entry multiStringArgument TwentyOne(void)
      output << "Starting TwentyOne ====>\n\n";

      GlobalMulStgArg.first = "First, changed by server.";
      GlobalMulStgArg.second = "Also second changed by server.";
      GlobalMulStgArg.dbl = 201.102;
      GlobalMulStgArg.last = "Even last changed by server.";

      output << "\nReturning from TwentyOne <====\n";

      return GlobalMulStgArg;
end;

// ------------------------------------------------------------------ //

entry memberStringArgument TwentyTwo(void)
      output << "Starting TwentyTwo ====>\n\n";

      GlobalMemStgArg.sa.st = "Set in TwentyTwo.";

      GlobalMemStgArg.st = "Modified in TwentyTwo.";

      GlobalMemStgArg.ma.first =
            "First, changed by server, in TwentyTwo.";

      GlobalMemStgArg.ma.second =
            "Also second changed by server, in TwentyTwo.";
```

```
      GlobalMemStgArg.ma.dbl = 2112.1221;
      GlobalMemStgArg.ma.last =
            "Even last changed by server, in TwentyTwo.";

      output << "\nReturning from TwentyTwo <====\n";

      return GlobalMemStgArg;
end;

// ------------------------------------------------------------------ //

baseMemberStringArgument GlobalBaseStgArg;
arrayStringArgument GlobalArrayArg;
arrayStringMember GlobalStgArray;

// ------------------------------------------------------------------ //

entry baseMemberStringArgument TwentyThree(void)
      output << "Starting TwentyThree ====>\n\n";


      output << "\nReturning from TwentyThree <====\n";

      return GlobalBaseStgArg;
end;

// ------------------------------------------------------------------ //

entry arrayStringArgument TwentyFour(void)
      output << "Starting TwentyFour ====>\n\n";


      output << "\nReturning from TwentyFour <====\n";

      return GlobalArrayArg;
end;

// ------------------------------------------------------------------ //

entry arrayStringMember TwentyFive(void)
      output << "Starting TwentyFive ====>\n\n";


      output << "\nReturning from TwentyFive <====\n";

      return GlobalStgArray;
end;

// ------------------------------------------------------------------ //

entry memberStringArgument TwentySix(multiStringArgument a,
            memberStringArgument b)

      output << "Starting TwentySix ====>\n\n";

      b.sa.st = "Set in TwentySix.";
```

```
      b.st = "Modified in TwentySix.";

      b.ma.first = "First, changed by server, in TwentySix.";
      b.ma.second = "Also second changed by server, in TwentySix.";
      b.ma.dbl = 2112.125;
      b.ma.last = "Even last changed by server, in TwentySix.";

      output << "\nReturning from TwentySix <====\n";

      return b;
end;

// ------------------------------------------------------------------ //

entry void Thirty(enumClassArgType e)
      output << "Starting Thirty ====>\n\n";

      output << e.t << '\n';
      output << [e.e] << '\n';

      if (e.e == _cherry) output << "It is Cherry!\n";
      else output << "Not Cherry!\n";
      endif;

      output << "\nReturning from Thirty <====\n";
end;

// ------------------------------------------------------------------ //

entry memberStringArgument ThirtyOne(remoteEnumes f,
            enumClassArgType e,
            memberStringArgument b,
            remoteEnumes l)

      output << "Starting ThirtyOne ====>\n\n";

      output << "First enum argument is: " << [f] << '\n';

      output << e.t << '\n';
      output << [e.e] << '\n';

      if (e.e == _cherry) output << "It is Cherry!\n";
      else output << "Not Cherry!\n";
      endif;

      b.sa.st = "Set in ThirtyOne.";
      b.st = "Modified in ThirtyOne.";
      b.ma.first = "First, changed by server, in ThirtyOne.";
      b.ma.second = "Also second changed by server, in ThirtyOne.";
      b.ma.dbl = 2112.125;
      b.ma.last = "Even last changed by server, in ThirtyOne.";

      output << "Last enum argument is: " << [l] << '\n';

      output << "\nReturning from ThirtyOne <====\n";

      return b;
```

```
end;

// ------------------------------------------------------------- //

entry enumClassArgType ThirtyTwo(memberStringArgument b,
            enumClassArgType e,
            remoteEnumes l)

      output << "Starting ThirtyTwo ====>\n\n";

      e.t = 313;
      e.e = _orange;

      output << "\nReturning from ThirtyTwo <====\n";

      return e;
end;

// ------------------------------------------------------------- //

entry remoteEnumes ThirtyThree(memberStringArgument m, remoteEnumes l)

      output << "Starting ThirtyThree ====>\n\n";

      output << m.sa.st << '\n';
      output << m.st << '\n';
      output << m.ma.first << '\n';
      output << m.ma.second << '\n';
      output << m.ma.dbl << '\n';
      output << m.ma.last << '\n';

      output << "\nReturning from ThirtyThree <====\n";

      return _cherry;
end;

// ------------------------------------------------------------- //

entry string ThirtyFour(string s, remoteEnumes e)

      output << "Starting ThirtyFour ====>\n\n";

      output << s << '\n';

      output << "\nReturning from ThirtyFour <====\n";

      return "I was returned by ThirtyFour.";

end;

// ------------------------------------------------------------- //
```

This is the program that loads RemoteComponent.zxe as component. We have specified **local** (the default) for load. This is really done so we can see the entire output on the local node. Otherwise, the output will be split between the local and the remote nodes.

The meaning of **local** load is that, RemoteComponent.zxe will be downloaded to the node where the following program is executing, and used as a component (its **entry** points are invoked). In contrast, **remote** specification means RemoteComponent.zxe will execute on the remote node, **serving as a component to the following program**.

```
//RemoteCallingModule.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////

#include"~/RemoteComponent.h"

////////////////////////////////////////////////////////////////////////
// The ip address must be to the node on which Z++ Internet Server is
// executing. The ~ is a placeholder for the correct path to reach the
// file, including drive name (like C:) if applicable.
////////////////////////////////////////////////////////////////////////

class Module = "Zpp://192.168.1.64/~/RemoteComponent.zxe"<local>

public:

      external void First(void);
      external void Second(void);
      external void Third(int);
      external void Fourth(bigArgument);
      external void Fifth(bigArgument, bigArgument);
      external int Sixth(void);
      external double Seventh(bigArgument);
      external returnObject Eighth(bigArgument);

      external void Nineth(stgArgument);
      external void Tenth(stgArgument, multiStringArgument);
      external void Eleventh(multiStringArgument,
                        stgArgument, memberStringArgument);

      external void Twelvth(derivedStringArgument);
      external void Thirteen(baseMemberStringArgument);

      external void Fourteen(baseMemberStringArgument,
                                          derivedStringArgument,
                                          memberStringArgument);

      external int Fifteen(string, baseMemberStringArgument,
                        string, string);

      external void Sixteen(arrayStringArgument);
```

```
      external void Seventeen(arrayStringMember);

// ------------------------------------------------------------ //

      external stgArgument Twenty(void);
      external multiStringArgument TwentyOne(void);
      external memberStringArgument TwentyTwo(void);
      external baseMemberStringArgument TwentyThree(void);
      external arrayStringArgument TwentyFour(void);
      external arrayStringMember TwentyFive(void);
      external memberStringArgument TwentySix(multiStringArgument,
                                              memberStringArgument);

// ------------------------------------------------------------ //

      external void Thirty(enumClassArgType);
      external memberStringArgument ThirtyOne(remoteEnumes,
                  enumClassArgType,
                  memberStringArgument,
                  remoteEnumes);

      external enumClassArgType ThirtyTwo(memberStringArgument,
                              enumClassArgType,
                              remoteEnumes);

      external remoteEnumes ThirtyThree(memberStringArgument,
                  remoteEnumes);

      external string ThirtyFour(string, remoteEnumes);

end;

////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "The calling module begins.\n";

      Module m; //create an instance

      output << "Main Module: Calling First.\n\n";

      m.First(); //call its entry Fisrt()

      output << "Main Module: Calling Second.\n\n";

      m.Second();

      output << "Main Module: Calling Third.\n\n";

      m.Third(97);

//create an instance to pass to remote call

      bigArgument ba(97, 9747, 47, 97.47);

      output << "Main Module: Calling Fourth.\n\n";
```

```
m.Fourth(ba);

bigArgument bb(717, 1329, 19, 273.051);

output << "Main Module: Calling Fifth.\n\n";

m.Fifth(bb, ba);

output << "Main Module: Calling Sixth.\n\n";

int intReturn = m.Sixth();
output << intReturn << '\n';

output << "Main Module: Calling Seventh.\n\n";

double dblReturn = m.Seventh(bb);
output << dblReturn << '\n';

returnObject rt;

output << "Main Module: Calling Eighth.\n\n";

rt = m.Eighth(ba);
output << rt.v << '\n';

output << "Main Module: Calling Nineth.\n\n";

stgArgument sa;
m.Nineth(sa);

output << "Main Module: Calling Tenth.\n\n";

multiStringArgument msa;
m.Tenth(sa, msa);

output << "Main Module: Calling Eleventh.\n\n";

memberStringArgument mma;
m.Eleventh(msa, sa, mma);

output << "Main Module: Calling Twelvth.\n\n";

derivedStringArgument dsa;
m.Twelvth(dsa);

output << "Main Module: Calling Thirteen.\n\n";

baseMemberStringArgument bmsa;
m.Thirteen(bmsa);

m.Fourteen(bmsa, dsa, mma);

output << "Main Module: Calling Fifteen.\n\n";

intReturn = m.Fifteen("First String Argument",
                  bmsa,
```

```
                        "Third argument is also string",
                        "Finally, last argument.");

      output << "Return is: " << intReturn << '\n';

      output << "Main Module: Calling Sixteen.\n\n";

      arrayStringArgument arrayArg;
      m.Sixteen(arrayArg);

      output << "Main Module: Calling Seventeen.\n\n";

      arrayStringMember arrayMem;
      m.Seventeen(arrayMem);

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% //

      sa = m.Twenty();
output << "Starting output after returning from Twenty ====>\n\n";

      output << sa.st << '\n';

output << "\nEnd of output after returning from Twenty ====>\n\n";

      msa = m.TwentyOne();

output << "Starting output after returning from TwentyOne ====>\n\n";

      output << msa.first << '\n';
      output << msa.second << '\n';
      output << msa.dbl << '\n';
      output << msa.last << '\n';

output << "\nEnd of output after returning from TwentyOne ====>\n\n";

      mma = m.TwentyTwo();

output << "Starting output after returning from TwentyTwo ====>\n\n";

      output << mma.sa.st << '\n';
      output << mma.st << '\n';
      output << mma.ma.first << '\n';
      output << mma.ma.second << '\n';
      output << mma.ma.dbl << '\n';
      output << mma.ma.last << '\n';

output << "\nEnd of output after returning from TwentyTwo ====>\n\n";

      output << "Main Module: Calling TwentyThree.\n\n";

      bmsa = m.TwentyThree();

      output << "Main Module: Calling TwentyFour.\n\n";

      arrayArg = m.TwentyFour();

      arrayMem = m.TwentyFive();
```

```
output << "Starting output after returning from TwentyFive ====>\n\n";

  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 2; j++)
      output << "Index i is : " << i << " and j is : " << j << '\n';
      output << arrayMem.asm[i][j] << '\n';
    endfor;
  endfor;

output << "\nEnd of output after returning from TwentyFive ====>\n\n";

      mma = m.TwentySix(msa, mma);

output << "Starting output after returning from TwentySix ====>\n\n";

      output << mma.sa.st << '\n';
      output << mma.st << '\n';
      output << mma.ma.first << '\n';
      output << mma.ma.second << '\n';
      output << mma.ma.dbl << '\n';
      output << mma.ma.last << '\n';

output << "\nEnd of output after returning from TwentySix ====>\n\n";

// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% //

      enumClassArgType ect;
      m.Thirty(ect);

      remoteEnumes reOne = _orange, reTwo = _apple;
      mma = m.ThirtyOne(reOne, ect, mma, reTwo);

output << "Starting output after returning from ThirtyOne ====>\n\n";

      output << mma.sa.st << '\n';
      output << mma.st << '\n';
      output << mma.ma.first << '\n';
      output << mma.ma.second << '\n';
      output << mma.ma.dbl << '\n';
      output << mma.ma.last << '\n';

output << "\nEnd of output after returning from ThirtyOne ====>\n\n";

      ect = m.ThirtyTwo(mma, ect, reTwo);

output << "Starting output after returning from ThirtyTwo ====>\n\n";

      output << ect.t << '\n';
      output << [ect.e] << '\n';

output << "\nEnd of output after returning from ThirtyTwo ====>\n\n";

      reTwo = m.ThirtyThree(mma, reTwo);

output << "Starting output after returning from ThirtyThree ====>\n\n";

      output << [reTwo] << '\n';
```

```
output << "\nEnd of output after returning from ThirtyThree ====>\n\n";

      string s = m.ThirtyFour("Hi there", reOne);
      output << s << '\n';

      output << "Main Module: Successfully terminating =====>>>>>\n\n";

end;
```

The output is as follows. The output is for **local** load. The output of **remote** load will be split between the local machine, and the remote node.

```
The calling module begins.
Main Module: Calling First.

First: Hello World!
Main Module: Calling Second.

Second: Hello World!
Main Module: Calling Third.

Third: the argument is 97
Main Module: Calling Fourth.

Starting Fourth ====>

bigArgument.n : 97
bigArgument.M.d : 97.47
bigArgument::leftBaseType::g : 9747
bigArgument::rightBaseType::t : 47

Returning from Fourth <====
Main Module: Calling Fifth.

Starting Fifth ====>

Printing First argument ====>

bigArgument.n : 717
bigArgument.M.d : 273.051
bigArgument::leftBaseType::g : 1329
bigArgument::rightBaseType::t : 19
Printing Second argument ====>

bigArgument.n : 97
bigArgument.M.d : 97.47
bigArgument::leftBaseType::g : 9747
bigArgument::rightBaseType::t : 47

Returning from Fifth <====
Main Module: Calling Sixth.
```

```
Starting Sixth ====>


Returning from Sixth <====
57
Main Module: Calling Seventh.

Starting Seventh ====>

Printing argument ====>

bigArgument.n : 717
bigArgument.M.d : 273.051
bigArgument::leftBaseType::g : 1329
bigArgument::rightBaseType::t : 19

Returning from Seventh <====
75.93
Main Module: Calling Eighth.

Starting Eighth ====>

Printing argument ====>

bigArgument.n : 97
bigArgument.M.d : 97.47
bigArgument::leftBaseType::g : 9747
bigArgument::rightBaseType::t : 47

Returning from Eighth <====
747
Main Module: Calling Nineth.

Starting Nineth ====>

Printing argument ====>

I am default string.

Returning from Nineth <====
Main Module: Calling Tenth.

Starting Tenth ====>

Printing argument ====>

I am default string.
I am first string.
I am second string.
456.123
I am last one.

Returning from Tenth <====
Main Module: Calling Eleventh.

Starting Eleventh ====>
```

```
Printing multiStringArgument ====>

I am first string.
I am second string.
456.123
I am last one.
Printing stgArgument ====>

I am default string.
Printing memberStringArgument ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.

Returning from Eleventh <====
Main Module: Calling Twelvth.

Starting Twelvth ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.

Returning from Twelvth <====
Main Module: Calling Thirteen.

Starting Thirteen ====>

Priting base derivedStringArgument ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.
Priting base memberStringArgument ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.
Priting member memberStringArgument ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
```

```
456.123
I am last one.
Priting member multiStringArgument ====>

I am first string.
I am second string.
456.123
I am last one.
Priting member derivedStringArgument ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.

Returning from Thirteen <====
Main Module: Calling Fifteen.

Starting Fifteen ====>

First the big argument ====>

Priting base derivedStringArgument ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.
Priting base memberStringArgument ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.
Priting member memberStringArgument ====>

I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.
Priting member multiStringArgument ====>

I am first string.
I am second string.
456.123
I am last one.
Priting member derivedStringArgument ====>

I am default string.
```

```
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.
Now three string arguments ====>

First String Argument
Third argument is also string
Finally, last argument.

Returning from Fifteen <====
Return is: 56
Main Module: Calling Sixteen.

Starting Sixteen ====>

Index is : 0
I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.
Index is : 1
I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.
Index is : 2
I am default string.
I am the self string member.
I am first string.
I am second string.
456.123
I am last one.

Returning from Sixteen <====
Main Module: Calling Seventeen.

Starting Seventeen ====>

Index i is : 0 and j is : 0
Joe
Index i is : 0 and j is : 1
Joe Smith
Index i is : 1 and j is : 0
Jim
Index i is : 1 and j is : 1
Jim Black
Index i is : 2 and j is : 0
Jack
Index i is : 2 and j is : 1
Jack White
Index i is : 3 and j is : 0
```

```
John
Index i is : 3 and j is : 1
John Brown

Returning from Seventeen <====
Starting Twenty ====>


Returning from Twenty <====
Starting output after returning from Twenty ====>

I have been changed by remote server.

End of output after returning from Twenty ====>

Starting TwentyOne ====>


Returning from TwentyOne <====
Starting output after returning from TwentyOne ====>

First, changed by server.
Also second changed by server.
201.102
Even last changed by server.

End of output after returning from TwentyOne ====>

Starting TwentyTwo ====>


Returning from TwentyTwo <====
Starting output after returning from TwentyTwo ====>

Set in TwentyTwo.
Modified in TwentyTwo.
First, changed by server, in TwentyTwo.
Also second changed by server, in TwentyTwo.
2112.12
Even last changed by server, in TwentyTwo.

End of output after returning from TwentyTwo ====>

Main Module: Calling TwentyThree.

Starting TwentyThree ====>


Returning from TwentyThree <====
Main Module: Calling TwentyFour.

Starting TwentyFour ====>


Returning from TwentyFour <====
Starting TwentyFive ====>
```

```
Returning from TwentyFive <====
Starting output after returning from TwentyFive ====>

Index i is : 0 and j is : 0
Joe
Index i is : 0 and j is : 1
Joe Smith
Index i is : 1 and j is : 0
Jim
Index i is : 1 and j is : 1
Jim Black
Index i is : 2 and j is : 0
Jack
Index i is : 2 and j is : 1
Jack White
Index i is : 3 and j is : 0
John
Index i is : 3 and j is : 1
John Brown

End of output after returning from TwentyFive ====>

Starting TwentySix ====>


Returning from TwentySix <====
Starting output after returning from TwentySix ====>

Set in TwentySix.
Modified in TwentySix.
First, changed by server, in TwentySix.
Also second changed by server, in TwentySix.
2112.13
Even last changed by server, in TwentySix.

End of output after returning from TwentySix ====>

Starting Thirty ====>

67
1
It is Cherry!

Returning from Thirty <====
Starting ThirtyOne ====>

First enum argument is: 2
67
1
It is Cherry!
Last enum argument is: 0

Returning from ThirtyOne <====
Starting output after returning from ThirtyOne ====>

Set in ThirtyOne.
```

```
Modified in ThirtyOne.
First, changed by server, in ThirtyOne.
Also second changed by server, in ThirtyOne.
2112.13
Even last changed by server, in ThirtyOne.

End of output after returning from ThirtyOne ====>

Starting ThirtyTwo ====>


Returning from ThirtyTwo <====
Starting output after returning from ThirtyTwo ====>

313
2

End of output after returning from ThirtyTwo ====>

Starting ThirtyThree ====>

Set in ThirtyOne.
Modified in ThirtyOne.
First, changed by server, in ThirtyOne.
Also second changed by server, in ThirtyOne.
2112.13
Even last changed by server, in ThirtyOne.

Returning from ThirtyThree <====
Starting output after returning from ThirtyThree ====>

1

End of output after returning from ThirtyThree ====>

Starting ThirtyFour ====>

Hi there

Returning from ThirtyFour <====
I was returned by ThirtyFour.
Main Module: Successfully terminating =====>>>>>
```

Section III.4.H.2. Remote loading of a Remote Component

The term module is conceptual. When a **class** is used as a representation of a component, we refer to it as a **module**. For instance, in previous section, in the program RemoteCallingModule.zpp, we have the following definition.

```
class Module = "Zpp://192.168.1.64/~/RemoteComponent.zxe"<local>
```

This is a **class** named Module. However, because of the assignment **operator** and the URL following the name of the **class**, it **represents a component**. In this case, the component is the executable of a Z++ program called RemoteComponent.zxe.

There are three scenarios in loading a component, which depend **solely** on the URL, and **without any change** to the rest of the program. Two of these scenarios are for the case when the program that loads a component, and the component reside on two different nodes (two different physical computers). In this case, we say that the component is remote, or we speak of a **remote component**.

A remote component can be loaded and executed as a component on the local node. The above definition of the **class** Module is an example of a **local load**. In order to execute a remote component remotely (a **remote load**), we only need to change the **local** specification to **remote**, as follows.

```
class Module = "Zpp://192.168.1.64/~/RemoteComponent.zxe"<remote>
```

The URL for a remote component will have the prefix `"Zpp://x.x.x.x/"` followed by full pathname to reach the file, where x.x.x.x is the IP address of Z++ Internet Server. The specification **local** is the **default**.

When a component resides on the same computer as the program loading it, the URL is just the full pathname of the component, and the Z++ Internet Server **has no role**.

Section III.4.I. Threads and Components

The example of this section illustrates simultaneous use of **task** thread, global threads and components.

The first program, CalledComponent.zpp is the component to be loaded by the second program, CallingComponent.zpp.

This is the component to be loaded. It simply provides a set of **entry** points with various signatures including fundamental types and structures.

```
// CalledComponent.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// This is intended to be loaded as component
// by "CallingComponent.zpp".
//--------------------------------------------------------------------//
// Types defined below are used as types of arguments
// and return objects.
//--------------------------------------------------------------------//

enum someEnums {_first, _second, _third};

//--------------------------------------------------------------------//

struct One
      int i;
      short s;
      char c;

      One(void);
end;

One::One(void)
      i = 75;
      s = 6;
      c = 'Y';
end;

struct Two
      someEnums e;
      string s;
      double d;

      Two(void);
end;

Two::Two(void)
      e = _second;
      s = "I am member of Two.";
```

```
      d = 71.35;
end;

struct Three
      int i;
      double d;

      Three(void);
end;

Three::Three(void)
       i = 17;
       d = 19.47;
end;

Two globalForReturn; // For returning a global object, as well.

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "I must be called from CallingComponent.zpp!\n";
end;

entry One FirstEntry(Three n)
      output << "FirstEntry. i is: " << n.i << " d is: " << n.d <<
'\n';
      One O;
      return O;
end;

entry double SecondEntry(string s)
      output << "SecondEntry. s is: " << s << '\n';
      return 47.11;
end;

entry short ThirdEntry(Two t)
      output << "ThirdEntry...\n";
      output << "t.e is: " << [t.e] << '\n';
      output << "t.s is: " << t.s << '\n';
      output << "t.d is: " << t.d << '\n';
      return 8;
end;

entry Two FourthEntry(someEnums e)
      output << "FourthEntry. e is: " << [e] << '\n';
      return globalForReturn;
end;
```

The following program loads the previous as component and invokes its **entry** points. While the program seems so simple, it is actually formidable, if not impossible to be written in any other language.

We have not used **invariants**, constraints, idler or signal handlers for the **task** only to keep the focus on the simultaneous use of threads, signals and invoking **entry** points of components.

The details of the program are explained as comments in the program.

```
// CallingComponent.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
// For this example first compile
// CalledComponent.zpp to CalledComponent.zxe.
//--------------------------------------------------------------------//
// Types defined below are used as types of
// arguments and return objects.
//--------------------------------------------------------------------//

enum someEnums {_first, _second, _third};

//--------------------------------------------------------------------//

struct One
      int i;
      short s;
      char c;

      One(void);
end;

One::One(void)
      i = 57;
      s = 15;
      c = 'U';
end;

struct Two
      someEnums e;
      string s;
      double d;

      Two(void);
end;

Two::Two(void)
      e = _third;
      s = "String member of Two.";
      d = 35.71;
end;
```

```
struct Three
      int i;
      double d;

      Three(void);
end;

Three::Three(void)
       i = 88;
       d = 47.97;
end;

////////////////////////////////////////////////////////////////////////
// The type constructor task can be used for defining
// a locally loaded module. Task is threaded.
//--------------------------------------------------------------------//
// It is not necessary to define a new set of methods
// for invoking entry points of a component.
// For instance, we do not need FirstMethod() to invoke the
// entry FirstEntry(). This style is shown as an illustration
// in case we need to use method constraints when calling
// entry points.
////////////////////////////////////////////////////////////////////////

task component = "~CalledComponent.zxe"

// These are the entry points of module CalledComponent.zpp

      external One FirstEntry(Three);
      external double SecondEntry(string);
      external short ThirdEntry(Two);
      external Two FourthEntry(someEnums);

public:

      One FirstMethod(Three);
      double SecondMethod(string);
      short ThirdMethod(Two);
      Two FourthMethod(someEnums);

end;

One component::FirstMethod(Three m)
      return FirstEntry(m);
end;

double component::SecondMethod(string s)
      return SecondEntry(s);
end;

short component::ThirdMethod(Two t)
      return ThirdEntry(t);
end;

Two component::FourthMethod(someEnums e)
      return FourthEntry(e);
end;
```

```
////////////////////////////////////////////////////////////////////////
// The following signals are used for thread communication.
//--------------------------------------------------------------------//

enum myPlainSignals : signalEventType {
      _SIGNAL_Thread_ended_1,
      _SIGNAL_Thread_ended_2,
      _SIGNAL_Thread_ended_3,
      _SIGNAL_Thread_ended_4,
      _SIGNAL_Thread_ended_5,
      _SIGNAL_Thread_ended_6
};

enum myEntireSignals : threadEntireEventType {
      _THREAD_ENTIRE_Signal_1
};

////////////////////////////////////////////////////////////////////////
// Below, four global threads are defined, each calling
// an entry point of the component as indicated.
// main() initiates the threads.
// Each thread waits until it receives the entire signal
// _THREAD_ENTIRE_Signal_1 from main() before calling the
// entry point.
// When the execution of entry is complete, each thread
// signals main() before terminating.
////////////////////////////////////////////////////////////////////////

//--------------------------------------------------------------------//
// Invokes "entry One FirstEntry(Three n)"

void globalThreadOne(component& T)<thread>
      accepts(_THREAD_ENTIRE_Signal_1)

// Wait for signal from main()

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

// Call entry point of component

      Three t;
      One n = T.FirstMethod(t);
      output << "globalThreadOne()...\n";
      output << "One.i is: " << n.i << '\n';
      output << "One.s is: " << n.s << '\n';
      output << "One.c is: " << n.c << '\n';

// Tell main() we are terminating.

      signal <- _SIGNAL_Thread_ended_1;
end;

//--------------------------------------------------------------------//
// Invokes "entry double SecondEntry(string s)"

void globalThreadTwo(component& T)<thread>
```

```
      accepts(_THREAD_ENTIRE_Signal_1)

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

      double d = T.SecondMethod("Hello from globalThreadTwo().");
      output << "globalThreadTwo(). d is: " << d << '\n';

      signal <- _SIGNAL_Thread_ended_2;
end;

//---------------------------------------------------------------------//
// Invokes "entry short ThirdEntry(Two t)"

void globalThreadThree(component& T)<thread>
      accepts(_THREAD_ENTIRE_Signal_1)

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

      Two t;
      short h = T.ThirdMethod(t);
      output << "globalThreadThree(). h is: " << h << '\n';

      signal <- _SIGNAL_Thread_ended_3;
end;

//---------------------------------------------------------------------//
// Invokes "entry Two FourthEntry(someEnums e)"

void globalThreadFour(component& T)<thread>
      accepts(_THREAD_ENTIRE_Signal_1)

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

      Two t = T.FourthMethod(_first);
      output << "globalThreadFour()...\n";
      output << "Two.e is: " << [t.e] << '\n';
      output << "Two.s is: " << t.s << '\n';
      output << "Two.d is: " << t.d << '\n';

      signal <- _SIGNAL_Thread_ended_4;
end;

/////////////////////////////////////////////////////////////////////
// Below are the definitions of two global threads.
// Each thread invokes two entry points, as indicated
// without waiting for signal from main().
// Before termination, each thread signals main().
/////////////////////////////////////////////////////////////////////

//---------------------------------------------------------------------//
// Invokes "entry One FirstEntry(Three n)" and
// "entry short ThirdEntry(Two t)"

void globalThreadOneThree(void)

      component X; // Create an instance of component
```

```
// Call entry points of component

      Three t;
      One n = X.FirstMethod(t);

      output << "globalThreadOneThree(). ";
      output << "After returning from FirstMethod()...\n";

      output << "One.i is: " << n.i << '\n';
      output << "One.s is: " << n.s << '\n';
      output << "One.c is: " << n.c << '\n';

      Two w;
      short s = X.ThirdMethod(w);

      output << "globalThreadOneThree(). ";
      output << "ThirdMethod() returned: " << s << '\n';

// Tell main() we are done.

      signal <- _SIGNAL_Thread_ended_5;
end;

//-------------------------------------------------------------------------//
// Invokes "entry double SecondEntry(string s)" and
// "entry Two FourthEntry(someEnums e)"

void globalThreadTwoFour(void)

      component X;

      double d = X.SecondMethod("Hello from globalThreadTwoFour().\n");
      output << "globalThreadTwoFour(). SecondMethod() returned: " << d
<< '\n';

      Two w = X.FourthMethod(_third);

      output << "globalThreadTwoFour(). ";
      output << "After calling FourthMethod()...\n";

      output << "Two.e is: " << [w.e] << '\n';
      output << "Two.s is: " << w.s << '\n';
      output << "Two.d is: " << w.d << '\n';

      signal <- _SIGNAL_Thread_ended_6;
end;

////////////////////////////////////////////////////////////////////////
//-------------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

// Create an instance of task component.

      component X;
```

```
// Make four global threads, and pass the component
// to them for invoking entry points.

      globalThreadOne(X);
      globalThreadTwo(X);
      globalThreadThree(X);
      globalThreadFour(X);

// Tell the four threads to begin calling entries

      signal <- _THREAD_ENTIRE_Signal_1;

// Make two global threads that declare their own instance
// of component.

      globalThreadOneThree();
      globalThreadTwoFour();

// Now, we also invoke all entry points of component X.

      Three t;
      One n = X.FirstMethod(t);
      output << "Main. After returning from FirstMethod()...\n";
      output << "One.i is: " << n.i << '\n';
      output << "One.s is: " << n.s << '\n';
      output << "One.c is: " << n.c << '\n';

      double d = X.SecondMethod("Hello from main.\n");
      output << "Main. SecondMethod() returned: " << d << '\n';

      Two w;
      short s = X.ThirdMethod(w);
      output << "Main. ThirdMethod() returned: " << s << '\n';

      w = X.FourthMethod(_third);
      output << "Main after calling FourthMethod()...\n";
      output << "Two.e is: " << [w.e] << '\n';
      output << "Two.s is: " << w.s << '\n';
      output << "Two.d is: " << w.d << '\n';

// Wait until all six threads tell us they are done.

      do enddo(signal ? _SIGNAL_Thread_ended_1);
      do enddo(signal ? _SIGNAL_Thread_ended_2);
      do enddo(signal ? _SIGNAL_Thread_ended_3);
      do enddo(signal ? _SIGNAL_Thread_ended_4);

      do enddo(signal ? _SIGNAL_Thread_ended_5);
      do enddo(signal ? _SIGNAL_Thread_ended_6);

// We are all done.

      output << "Good-bye World!\n";
end;
```

Output is as follows.

The output is very hard to understand. Several threads simultaneously invoke the entry points and their outputs are intermixed. However, the clarity of output was not the goal.

```
Hello World!
SecondEntry. s is: Hello from globalThreadTwo().
FourthEntry. e is: globalThreadTwo(). d is: 47.11FirstEntry. i is: 88 d
is: 0
47.97

globalThreadFour()...
FirstEntry. i is: Two.e is: 1globalThreadOneThree(). After returning
from FirstMethod()...
One.i is: 88 d is: 47.97
Two.s is: 75

I am member of Two.
Two.d is: One.s is: 671.35

One.c is: Y
globalThreadOne()...
One.i is: ThirdEntry...
t.e is: ThirdEntry...
t.e is: 75
22
One.s is: 6
t.s is: String member of Two.t.s is: String member of Two.

One.c is: Y
t.d is: t.d is: 35.71
35.71

globalThreadOneThree(). ThirdMethod() returned: globalThreadThree(). h
is: 8
8
SecondEntry. s is: Hello from globalThreadTwoFour().

globalThreadTwoFour(). SecondMethod() returned: 47.11
FourthEntry. e is: 2
globalThreadTwoFour(). After calling FourthMethod()...
Two.e is: 1
Two.s is: I am member of Two.
Two.d is: 71.35
FirstEntry. i is: 88 d is: 47.97
Main. After returning from FirstMethod()...
One.i is: 75
One.s is: 6
One.c is: Y
SecondEntry. s is: Hello from main.

Main. SecondMethod() returned: 47.11
ThirdEntry...
```

```
t.e is: 2
t.s is: String member of Two.
t.d is: 35.71
Main. ThirdMethod() returned: 8
FourthEntry. e is: 2
Main after calling FourthMethod()...
Two.e is: 1
Two.s is: I am member of Two.
Two.d is: 71.35
Good-bye World!
```

Chapter III.5. Linking with other languages

A system language, such as C++, creates system programs. System programs are aware of system calls and can handle time-sensitive situations, among other things. Z++ is an abstraction for developing applications, independent of the internals of a computing device. A system language cannot provide the facilities of Z++ for creating modern applications. On the other hand, at times the ability to inter-operate with a system program has practical uses, like when time-sensitive actions are required.

Z++ inter-operability with system languages is provided via linkage mechanism with dynamic libraries of those languages. Thus. Z++ interoperability **only requires the ability of building dynamic libraries** from (the compiler of) a system language. The form of dynamic library is not relevant. For instance, Z++ can link with Windows DLL, and Loadable Modules on UNIX platforms.

The Z++ mechanism for linking with dynamic libraries of system programs is **module**. In order to distinguish between Z++ native modules and modules that link with system program, we refer to the latter kind as **Linkage Modules**, rather than plain **modules**, or Z++ native modules.

Linkage module can only use class, while native modules can use class and task.

You can link with a **static** library in target language, or a **dynamic** library. In both cases the Z++ compiler generates a **source** file in target language. You then build a dynamic library in the target language using the generated source file, which will be used by Z47 Processor to direct the calls into the library of the target language.

We will use C++ as the target language for illustrations.

Section III.5.A. Static Linkage Modules

We begin with an example. Consider the following Z++ program.

```
// MathFunctions.zpp

#include<iostream.h>

// ------------------------------------------------------------------ //

namespace mathSpace

private: // Make our work private, only export final functions

      struct MathFunctions "C++" static = "MathLibrary"

            library "math.h";

            external double cos(double);
            external double sin(double);
            external double tan(double);

            external double acos(double);
            external double asin(double);
            external double atan(double);

            external double cosh(double);
            external double sinh(double);
            external double tanh(double);

            external double exp(double);
            external double log(double);
            external double log10(double);

      end;

      MathFunctions MathObject;

// ------------------------------------------------------------------ //
// Only the following global functions are exported, and accessible.
// The names of these functions are same as they are in C and C++.
// ------------------------------------------------------------------ //

public:

      double cos(double x) return MathObject.cos(x); end;
      double sin(double x) return MathObject.sin(x); end;
      double tan(double x) return MathObject.tan(x); end;

      double acos(double x) return MathObject.acos(x); end;
      double asin(double x) return MathObject.asin(x); end;
      double atan(double x) return MathObject.atan(x); end;

      double cosh(double x) return MathObject.cosh(x); end;
      double sinh(double x) return MathObject.sinh(x); end;
```

```
        double tanh(double x) return MathObject.tanh(x); end;

        double exp(double x) return MathObject.exp(x); end;
        double log(double x) return MathObject.log(x); end;
        double log10(double x) return MathObject.log10(x); end;

endspace;

// ------------------------------------------------------------------ //

const double pi = 3.1415;

// ------------------------------------------------------------------ //

entry void main(void)

        using namespace ioSpace;
        using namespace mathSpace;

        output << atan(cos(pi/3)) << '\n';
        output << sin(pi/4) << '\n';

        output << sinh(pi/3) << '\n';

        output << log(1) << '\n';
        output << exp(3) << '\n';

end;
```

The output of this program is as follows.

```
0.463669
0.70709
1.24932
0
20.0855
```

Remark. The use of **namespace** is unnecessary. It is only used so C++ library functions can be called without the use of an object, as they are usually called.

The **static** linkage module MathFunctions is defined via a **struct**. Right after the name of **struct** comes the specification "C++", which tells the Z++ compiler that we going to link with a C++ library.

Next, the specification **static** means that we are going to link with a static library of C++.

The assignment operator is used the same way as for Z++ native modules. It simply means that we are defining a **module**.

The string following the assignment operator is full path-filename of the source file where we want the Z++ compiler to **generate** the source file in target language. The

filename must not have extension. Z++ Compiler will add the extension. For instance, it may look like: **"C:/directory/filename"**, but **not** like: **"C:/directory/filename.cpp"**.

The Z++ compiler will generate a source file in the folder "directory" named "filename.cpp". Using a C++ compiler, build a DLL from this source. Then copy the DLL in same folder, in this case: **"C:/diretory/filename.dll"**. This is the DLL that Z47 will load at execution time for sending calls to C++ static math library.

Remark. It is important **not** to include file-extension, such as ".cpp". The Z++ Compiler will **add** the extension and this may result in confusion.

The **library** statement takes a list of comma separated header files that are needed to be included in the source file that Z++ Compiler generates. For instance in this case, when building the DLL with a C++ compiler, the header file "math.h" needs to be included.

When there are no header files to include, you can leave out the **library** statement. However, for documentation purposes, you can use the **library** statement without operands, indicating that no header files need to be included in the generated file, as follows.

```
library;
```

The **external** methods are same as for Z++ native modules. The prototypes must match those in the static library of the target language. Z++ Compiler generates the bodies of **external** methods, as ever.

Section III.5.A.1. Reuse through Z++ Modules

A static linkage module can be made wrapped with a Z++ native module for reuse in other Z++ programs. Though straightforward, we illustrate this with an example.

The following program is the Z++ wrapper. We will use the same DLL built in previous project. Just make sure the string after assignment, "MathLibrary" is the correct path to reach it where it resides currently, or wherever you copy it for use with this project. Otherwise Z47 will not find the DLL and you will get an exception.

```
// MathFunctions.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

struct MathFunctions "C++" static = "MathLibrary"

      library "math.h";

      external double cos(double);
```

```
        external double sin(double);
        external double tan(double);

        external double acos(double);
        external double asin(double);
        external double atan(double);

        external double cosh(double);
        external double sinh(double);
        external double tanh(double);

        external double exp(double);
        external double log(double);
        external double log10(double);


end;

// ------------------------------------------------------------------ //

MathFunctions MathObject;

const double pi = 3.1415;

// ------------------------------------------------------------------ //

entry void main(void)
        output << "C++ static math library.\n";
end;

entry double cos(double x)
        return MathObject.cos(x);
end;

entry double sin(double x)
        return MathObject.sin(x);
end;

entry double tan(double x)
        return MathObject.tan(x);
end;

entry double acos(double x)
        return MathObject.acos(x);
end;

entry double asin(double x)
        return MathObject.asin(x);
end;

entry double atan(double x)
        return MathObject.atan(x);
end;

entry double cosh(double x)
        return MathObject.cosh(x);
end;
```

```
entry double sinh(double x)
      return MathObject.sinh(x);
end;

entry double tanh(double x)
      return MathObject.tanh(x);
end;

entry double exp(double x)
      return MathObject.exp(x);
end;

entry double log(double x)
      return MathObject.log(x);
end;

entry double log10(double x)
      return MathObject.log10(x);
end;
```

The above component can now be used in any Z++ program. Below is an example. As you see, this program is identical to the example of previous section with minor differences. Since we are going to load a native Z++ module, there are no specifications "C++" and **static**. Furthermore, "MathFunctions.zxe" is full-path name of Z++ executable that we built just before this.

```
// Sample.zpp

#include<iostream.h>

// ------------------------------------------------------------------ //

namespace mathSpace

private:

      struct MathFunctions = "MathFunctions.zxe"

            external double cos(double);
            external double sin(double);
            external double tan(double);

            external double acos(double);
            external double asin(double);
            external double atan(double);

            external double cosh(double);
            external double sinh(double);
            external double tanh(double);

            external double exp(double);
            external double log(double);
            external double log10(double);
```

```
        end;

        MathFunctions MathObject;

public:

        double cos(double x) return MathObject.cos(x); end;
        double sin(double x) return MathObject.sin(x); end;
        double tan(double x) return MathObject.tan(x); end;

        double acos(double x) return MathObject.acos(x); end;
        double asin(double x) return MathObject.asin(x); end;
        double atan(double x) return MathObject.atan(x); end;

        double cosh(double x) return MathObject.cosh(x); end;
        double sinh(double x) return MathObject.sinh(x); end;
        double tanh(double x) return MathObject.tanh(x); end;

        double exp(double x) return MathObject.exp(x); end;
        double log(double x) return MathObject.log(x); end;
        double log10(double x) return MathObject.log10(x); end;

endspace;

// ------------------------------------------------------------------ //

const double pi = 3.1415;

// ------------------------------------------------------------------ //

entry void main(void)

        using namespace ioSpace;
        using namespace mathSpace;

        output << atan(cos(pi/3)) << '\n';
        output << sin(pi/4) << '\n';

        output << sinh(pi/3) << '\n';

        output << log(1) << '\n';
        output << exp(3) << '\n';

end;
```

The output of this program is as follows.

```
0.463669
0.70709
1.24932
0
20.0855
```

Section III.5.B. Dynamic Linkage Modules

There times that dynamic libraries in target language already exist for other reasons. Z++ can also link with existing dynamic libraries of other languages.

The process of linking with dynamic libraries is the same as linking with static libraries. Z++ Compiler generates a source file in the target language. Using the generated file, you build a dynamic library using a compiler for the target language. Finally, you copy the dynamic library you just built at location you indicated in your Z++ program.

In order to illustrate linking with dynamic libraries, we make three DLLs in this section. However, note that we are only doing this to supply the existing dynamic libraries to link with. In practice, the libraries will already be available, perhaps for other uses.

Below are three C++ source files for building three C++ DLLs. The three files could be put into a single file, and make one DLL. Here, we wish to illustrate multiple-inheritance.

```
// TrigFuns.cpp

#include<math.h>

extern"C" _declspec (dllexport)
double ZppCos(double x) {return cos(x);}

extern"C" _declspec (dllexport)
double ZppSin(double x) {return sin(x);}

extern"C" _declspec (dllexport)
double ZppTan(double x) {return tan(x);}

extern"C" _declspec (dllexport)
double ZppAcos(double x) {return acos(x);}

extern"C" _declspec (dllexport)
double ZppAsin(double x) {return asin(x);}

extern"C" _declspec (dllexport)
double ZppAtan(double x) {return atan(x);}

// HyperFuns.cpp

#include<math.h>

extern"C" _declspec (dllexport)
double ZppCosh(double x) {return cosh(x);}

extern"C" _declspec (dllexport)
double ZppSinh(double x) {return sinh(x);}

extern"C" _declspec (dllexport)
double ZppTanh(double x) {return tanh(x);}
```

```
// LogFuns.cpp

#include<math.h>

extern"C" _declspec (dllexport)
double ZppExp(double x) {return exp(x);}

extern"C" _declspec (dllexport)
double ZppLog(double x) {return log(x);}

extern"C" _declspec (dllexport)
double ZppLog10(double x) {return log10(x);}
```

Suppose, using a C++ compiler, these files are compiled to C++ DLLs with names `TrigLib.dll`, `HyperLib.dll` and `LogLib.dll`, respectively.

.

Section III.5.B.1 Example for Dynamic Linkage Modules

Now we write a Z++ program that will link with the DLLs of the previous section.

There are three linkage modules in the Z++ program listed below, one for each DLL of the previous section. We describe the linkage module `TrigFunctions` here. The reason for using three existing DLLs is to illustrate the fact that you can use multiple-inheritance with regard to external modules.

First, you notice the lack of specification **static**. When you leave out **static** you are telling the Z++ Compiler that you are going to link with existing dynamic libraries, in the target language.

The literal string following the assignment is full-pathname of the DLL, **including** its extension ".dll". In contrast, for static linkage modules this was name of source for the Z++ Compiler to generate. For dynamic linkage modules, the name of source is given to the compiler via **generated** statement, discussed next.

For **static** linking you used the **library** statement. For dynamic linking, instead you use the **`generated`** statement. The Compiler generates a C++ source file from the definition of a linkage module. It needs to know where to store the generated file. The literal string is full path to the location for storing the file, and the name of the file **without extension**. In this example the generated file will be named `"TrigFunctions.cpp"`. Note that the extension cpp is, added by the compiler, depending on the language specification like `"C++"`. However, except for the file, the Compiler expects all directories in the path to exist.

The **`external`** methods are prototypes of **exported** functions of the target dynamic library. As for static linkage modules, you only specify the prototypes for **`external`** methods and the Compiler generates their bodies.

Compiling this program generates three C++ source files named `TrigFunctions.cpp`, `HyperFunctions.cpp`, and `LogFunctions.cpp`, which we discuss in the next section.

```
// MathFunctions.zpp

#include <iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //
// Trig functions.
// ------------------------------------------------------------------ //

class TrigFunctions "C++" = "~/TrigLib.dll"

      generated "~/TrigFunctions";

protected:
```

```
      external double ZppCos(double);
      external double ZppSin(double);
      external double ZppTan(double);

      external double ZppAcos(double);
      external double ZppAsin(double);
      external double ZppAtan(double);
end;

// ------------------------------------------------------------------ //
// Hyperbolic functions.
// ------------------------------------------------------------------ //

class HyperFunctions "C++" = "~/HyperLib.dll"

      generated "~/HyperFunctions";

protected:

      external double ZppCosh(double);
      external double ZppSinh(double);
      external double ZppTanh(double);
end;

// ------------------------------------------------------------------ //
// Logarithm and exponential functions.
// ------------------------------------------------------------------ //

class LogFunctions "C++" = "~/LogLib.dll"

      generated "~/LogFunctions";

protected:

      external double ZppExp(double);
      external double ZppLog(double);
      external double ZppLog10(double);
end;

// ------------------------------------------------------------------ //

class MathFunctions : TrigFunctions, HyperFunctions, LogFunctions

public:

      double cos(double);
      double sin(double);
      double tan(double);

      double acos(double);
      double asin(double);
      double atan(double);

      double cosh(double);
      double sinh(double);
      double tanh(double);
```

```
	double exp(double);
	double log(double);
	double log10(double);

end;

// ------------------------------------------------------------- //

double MathFunctions::cos(double x) return ZppCos(x); end;
double MathFunctions::sin(double x) return ZppSin(x); end;
double MathFunctions::tan(double x) return ZppTan(x); end;

double MathFunctions::acos(double x) return ZppAcos(x); end;
double MathFunctions::asin(double x) return ZppAsin(x); end;
double MathFunctions::atan(double x) return ZppAtan(x); end;

double MathFunctions::cosh(double x) return ZppCosh(x); end;
double MathFunctions::sinh(double x) return ZppSinh(x); end;
double MathFunctions::tanh(double x) return ZppTanh(x); end;

double MathFunctions::exp(double x) return ZppExp(x); end;
double MathFunctions::log(double x) return ZppLog(x); end;
double MathFunctions::log10(double x) return ZppLog10(x); end;

// ------------------------------------------------------------- //

const double pi = 3.1415;

////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////

entry void main(void)

	MathFunctions MF;

	output << MF.atan(MF.cos(pi/3)) << '\n';
	output << MF.sin(pi/4) << '\n';

	output << MF.sinh(pi/3) << '\n';

	output << MF.log(1) << '\n';
	output << MF.exp(3) << '\n';

end;
```

The output of this program is as follows.

```
0.463669
0.70709
1.24932
0
20.0855
```

Section III.5.B.2 Sample of a generated source

Three source files `TrigFunctions.cpp`, `HyperFunctions.cpp`, and `LogFunctions.cpp` were generated in precious section. We use a C++ compiler and turn these source files into DLLs.

It is important to name the DLLs in accordance to the names of their source files. Z++ Compiler assumes the names will be `TrigFunctions.dll`, `HyperFunctions.dll`, and `LogFunctions.dll`. Z47 will look for them by those names using the path that you provided in **`generated`** statement. That is, after building the DLLs you will need to **copy** them to the **directory** specified by **`generated`** statement. This completes the process of creating linkage modules.

Here we list an example of the generated file `TrigFunctions.cpp` in case you would like to know how the mechanism works. There is no need to understand the program. It is generated by, one compiler for use by another compiler. Note that the word processor has broken long lines into multiple lines. But since this is a generated file we are not editing it.

```
// C:/Program Files/ZHMICRO/Z++
Visual/Examples/C++Linkage/MathLibrary/Generated/TrigFunctions.cpp
//               Generated by Z++ Linker

// ====> Make a dynamic library using this file and copy the library to
this directory <===

#include<windows.h>

void* ZppInterfaceReturn;

char ZppInterfaceReturnChar;
short ZppInterfaceReturnShort;
int ZppInterfaceReturnInt;
long ZppInterfaceReturnLong;
bool ZppInterfaceReturnBoolean;
float ZppInterfaceReturnFloat;
double ZppInterfaceReturnDouble;
unsigned char ZppInterfaceReturnUnsignedChar;
unsigned short ZppInterfaceReturnUnsignedShort;
unsigned int ZppInterfaceReturnUnsignedInt;
unsigned long ZppInterfaceReturnUnsignedLong;

char* ZppInterfaceReturnCharPointer;
short* ZppInterfaceReturnShortPointer;
int* ZppInterfaceReturnIntPointer;
long* ZppInterfaceReturnLongPointer;
bool* ZppInterfaceReturnBooleanPointer;
float* ZppInterfaceReturnFloatPointer;
double* ZppInterfaceReturnDoublePointer;
unsigned char* ZppInterfaceReturnUnsignedCharPointer;
unsigned short* ZppInterfaceReturnUnsignedShortPointer;
```

```
unsigned int* ZppInterfaceReturnUnsignedIntPointer;
unsigned long* ZppInterfaceReturnUnsignedLongPointer;

void* ZppInterfaceReturnVoidPointer;

HINSTANCE
hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary
_C__MathLibrary_Library_dll;

typedef double
(*C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppCos_Type)(double);
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppCos_Type
      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppCos;

typedef double
(*C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppSin_Type)(double);
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppSin_Type
      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppSin;

typedef double
(*C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppTan_Type)(double);
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppTan_Type
      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppTan;

typedef double
(*C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppAcos_Type)(double);
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppAcos_Type
      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppAcos;

typedef double
(*C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppAsin_Type)(double);
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppAsin_Type
      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppAsin;

typedef double
(*C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppAtan_Type)(double);
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppAtan_Type
      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppAtan;
```

```
extern"C" _declspec (dllexport) void* ZppExportedInterfaceFunction(int
fun, void* args[]) {

      switch(fun) {
            case 0:
                  ZppInterfaceReturnDouble =
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppCos(
                        *((double*)args[0]));
                  ZppInterfaceReturn = (void*)
&ZppInterfaceReturnDouble;
                  break;

            case 1:
                  ZppInterfaceReturnDouble =
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppSin(
                        *((double*)args[0]));
                  ZppInterfaceReturn = (void*)
&ZppInterfaceReturnDouble;
                  break;

            case 2:
                  ZppInterfaceReturnDouble =
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppTan(
                        *((double*)args[0]));
                  ZppInterfaceReturn = (void*)
&ZppInterfaceReturnDouble;
                  break;

            case 3:
                  ZppInterfaceReturnDouble =
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppAcos(
                        *((double*)args[0]));
                  ZppInterfaceReturn = (void*)
&ZppInterfaceReturnDouble;
                  break;

            case 4:
                  ZppInterfaceReturnDouble =
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppAsin(
                        *((double*)args[0]));
                  ZppInterfaceReturn = (void*)
&ZppInterfaceReturnDouble;
                  break;

            case 5:
                  ZppInterfaceReturnDouble =
C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C__
MathLibrary_Library_ZppAtan(
                        *((double*)args[0]));
                  ZppInterfaceReturn = (void*)
&ZppInterfaceReturnDouble;
```

```
                break;

            default: ZppInterfaceReturn = 0;
      }

      return ZppInterfaceReturn;
}

bool initialize(void) {
      hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathL
ibrary_C__MathLibrary_Library_dll =
            LoadLibrary("C:/Program Files/ZHMICRO/Z++
Visual/Examples/C++Linkage/MathLibrary/C++MathLibrary/TrigLib.dll");
      if
(!hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_dll) return false;

      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppCos =

      (C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibr
ary_C__MathLibrary_Library_ZppCos_Type)

      GetProcAddress(hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C
__Linkage_MathLibrary_C__MathLibrary_Library_dll, "ZppCos");
      if
(!C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppCos) return false;

      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppSin =

      (C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibr
ary_C__MathLibrary_Library_ZppSin_Type)

      GetProcAddress(hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C
__Linkage_MathLibrary_C__MathLibrary_Library_dll, "ZppSin");
      if
(!C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppSin) return false;

      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppTan =

      (C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibr
ary_C__MathLibrary_Library_ZppTan_Type)

      GetProcAddress(hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C
__Linkage_MathLibrary_C__MathLibrary_Library_dll, "ZppTan");
      if
(!C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppTan) return false;

      C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppAcos =
```

```
	(C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibr
ary_C__MathLibrary_Library_ZppAcos_Type)

	GetProcAddress(hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C
__Linkage_MathLibrary_C__MathLibrary_Library_dll, "ZppAcos");
	if
(!C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppAcos) return false;

	C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppAsin =

	(C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibr
ary_C__MathLibrary_Library_ZppAsin_Type)

	GetProcAddress(hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C
__Linkage_MathLibrary_C__MathLibrary_Library_dll, "ZppAsin");
	if
(!C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppAsin) return false;

	C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibra
ry_C__MathLibrary_Library_ZppAtan =

	(C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibr
ary_C__MathLibrary_Library_ZppAtan_Type)

	GetProcAddress(hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C
__Linkage_MathLibrary_C__MathLibrary_Library_dll, "ZppAtan");
	if
(!C__Program_Files_ZHMICRO_Z___Visual_Examples_C__Linkage_MathLibrary_C
__MathLibrary_Library_ZppAtan) return false;

	return true;
}

bool cleanup(void) {
	FreeLibrary(hdl_C__Program_Files_ZHMICRO_Z___Visual_Examples_C__L
inkage_MathLibrary_C__MathLibrary_Library_dll);
	return true;
}

bool WINAPI DllMain(HINSTANCE hinst, DWORD Reason, LPVOID Res) {
	switch(Reason) {
		case DLL_PROCESS_ATTACH:
		return initialize();
		case DLL_PROCESS_DETACH:
		return cleanup();
	}
	return true;
}
```

Section III.5.C. Linkage Module Options

Ordinarily, Z47 loads the generated dynamic library (static or dynamic linkage) just before calling an **external** method, and unloads the library right after the call returns. This may be useful when making an occasional call to the methods of a large linkage module.

It is possible to tell Z47 to load the dynamic library at the point of declaring an instance of it, and to unload the library at the destruction of the instance, like when the instance goes out of scope. This is done by, specifying the linkage module as **persistent**.

The specification **persistent** can be used with or without **static**, as shown below. The order of specifications **static** and **persistent** is **not** significant.

```
class ModuleName "C++" static persistent = "path-name"
```

or,

```
class ModuleName "C++" persistent = "path-name"
```

We may need to make changes to the Z++ source without affecting the calls to the C++ library. For instance in our example it is unlikely that we will change the type of parameters to the calls or the names of **external** functions. In such cases, there is no need to **regenerate** the C++ source files, even though we will **not rebuild** the DLLs (there was no change to require a rebuild).

More specifically, a relevant change that will require regeneration of C++ source files, and the rebuilding of the DLL is change of signature or the **return** of an **external** method, removal or addition of an **external** method, changing the **order** of **external** methods, or finally changes in the paths of files.

Remark. Note that changing the order of declaration of **external** methods of a linkage module is used in the source file generated by the Z++ Compiler. Thus, changing the order of **external** methods requires the regeneration of the source file, and rebuilding the dynamic library associated with it.

The specification **generated** (not to be confused with **generated statement**) is used to tell the Z++ Compiler **Not** to regenerate the source file. Like **persistent** the specification **generated** is optional, and can be used for static and dynamic linkage. The order of, **static**, **persistent** and **generated** is not significant.

```
class ModuleName "C++" static persistent generated = "path-name"
```

or,

```
class ModuleName "C++" persistent generated = "path-name"
```

Section III.5.D. A namespace example

The Z++ source file `MathFunctions.zpp` that we used for dynamic Linkage Module uses objects for calling math library functions, as in `MF.atan(MF.cos(pi/3))`. We can avoid the use of objects in math expression via the mechanism of **namespace**.

The following example can replace `MathFunctions.zpp`, leaving everything else unchanged (no need to rebuild any of the DLLs). The three namespaces `trigSpace`, `hyperSpace`, and `logSpace` are specified **`protected`** so they can only be used in a derivation by another **namespace**. Then **namespace** `mathSpace` inherits the three namespaces. In function main() we simply use the usual math library functions without reference to an object.

```
// MathFunctions.zpp

#include <iostream.h>

// ------------------------------------------------------------------ //
// Trig functions.
// ------------------------------------------------------------------ //

protected namespace trigSpace

struct TrigFunctions "C++" persistent = "~/TrigLib.dll"

      generated "~/TrigFunctions";

      external double ZppCos(double);
      external double ZppSin(double);
      external double ZppTan(double);

      external double ZppAcos(double);
      external double ZppAsin(double);
      external double ZppAtan(double);
end;

endspace;

// ------------------------------------------------------------------ //
// Hyperbolic functions.
// ------------------------------------------------------------------ //

protected namespace hyperSpace

struct HyperFunctions "C++" persistent  = "~/HyperLib.dll"

      generated "~/HyperFunctions";

      external double ZppCosh(double);
      external double ZppSinh(double);
      external double ZppTanh(double);
end;
```

```
endspace;

// ------------------------------------------------------------------ //
// Logarithm and exponential functions.
// ------------------------------------------------------------------ //

protected namespace logSpace

struct LogFunctions "C++" persistent  = "~/LogLib.dll"

      generated "~/LogFunctions";

      external double ZppExp(double);
      external double ZppLog(double);
      external double ZppLog10(double);
end;

endspace;

// ------------------------------------------------------------------ //
// We will include all our namespaces into one, mathSpace.
// Furthermore, we shall make the type MathFunctions and its
// instance MathObject private.
// In the public section of the namespace mathSpace we will define
// our math functions as global function within the namespace,
// and with same name as in C++ math library.
// ------------------------------------------------------------------ //

namespace mathSpace : trigSpace, hyperSpace, logSpace

private: // Make our work private, only export final functions

      struct MathFunctions : TrigFunctions, HyperFunctions,
                             LogFunctions
      end;

      MathFunctions MathObject;

// ------------------------------------------------------------------ //
// Only the following global functions are exported, and
// accessible. The names of these functions are same as
// they are in C and C++.
// ------------------------------------------------------------------ //

public:

      double cos(double x) return MathObject.ZppCos(x); end;
      double sin(double x) return MathObject.ZppSin(x); end;
      double tan(double x) return MathObject.ZppTan(x); end;

      double acos(double x) return MathObject.ZppAcos(x); end;
      double asin(double x) return MathObject.ZppAsin(x); end;
      double atan(double x) return MathObject.ZppAtan(x); end;

      double cosh(double x) return MathObject.ZppCosh(x); end;
      double sinh(double x) return MathObject.ZppSinh(x); end;
```

```
        double tanh(double x) return MathObject.ZppTanh(x); end;

        double exp(double x) return MathObject.ZppExp(x); end;
        double log(double x) return MathObject.ZppLog(x); end;
        double log10(double x) return MathObject.ZppLog10(x); end;

endspace;

// ------------------------------------------------------------------ //

const double pi = 3.1415;

// ------------------------------------------------------------------ //

entry void main(void)

        using namespace ioSpace;
        using namespace mathSpace;

        output << atan(cos(pi/3)) << '\n';
        output << sin(pi/4) << '\n';

        output << sinh(pi/3) << '\n';

        output << log(1) << '\n';
        output << exp(3) << '\n';

end;
```

Section III.5.E. Linkage Data Types

Data types that can be used for communication between a Z++ and a C++ program is sufficient for all needs. All numeric types from **char** to **double**, signed and unsigned, and **boolean** types are supported. In addition, pointers to these types and arrays of instance of these types, as well as Z++ **string**s as null-terminated C strings can be exchanged between Z++ and C++. Finally, **pass by reference** is also supported for linkage modules.

We illustrate the above with an example. We start with a C++ source file. We are going to make a Z++ dynamic linkage module that will link with the C++ DLL corresponding to the following C++ source file. The signature of the exported functions includes all **combinations of data types we mentioned, pointers, arrays and pass by reference**.

```
// CppSample.cpp

#include<windows.h>
#include<iostream>
using namespace std;

////////////////////////////////////////////////////////////////////////

extern"C" _declspec (dllexport)
double SimpleMethod(int one, double two, short three) {
      cout << "SimpleMethod received: \n";
      cout << "int: " << one;
      cout << " double: " << two;
      cout << " short: " << three << endl;
      two += one;
      two += three;
      cout << "SimpleMethod returning (double): " << two << endl;
      return two;
}

//--------------------------------------------------------------------//

extern"C" _declspec (dllexport)
unsigned long UnsignedMethod(unsigned int ui, long l, unsigned char uc)
{
      cout << "UnsignedMethod received:\n";
      cout << "unsigned int: " << ui << endl;
      cout << "long: " << l << endl;
      cout << "unsigned char: " << uc << endl;
      unsigned long ulr = ui * uc;
      cout << "UnsignedMethod returning (unsigned long): " << ulr <<
endl;
      return ulr;
}

//--------------------------------------------------------------------//

extern"C" _declspec (dllexport)
int* PointerMethod(int*** ippp, char* cp) {
      cout << "PointerMethod received:\n";
```

```
	cout << "int***: " << ***ippp << endl;
	cout << "char*: " << cp << endl;
	cout << "PointerMethod returning (int*)." << endl;
	return **ippp;
}

//-------------------------------------------------------------------//

extern"C" _declspec (dllexport)
void BooleanMethod(bool b, char** cpp) {
	cout << "BooleanMethod received boolean and it ";
	if (b) cout << "was true.\n";
	else cout << "was false.\n";
	cout << "BooleanMethod also received string : " << *cpp << endl;
}

//-------------------------------------------------------------------//

extern"C" _declspec (dllexport)
char* StringMethod(char* cpp) {
	cout << "StringMethod returning string : " << cpp << endl;
	return cpp;
}

//-------------------------------------------------------------------//

extern"C" _declspec (dllexport)
void ReferenceMethod(const unsigned int& uir, long*& lp, double& dr) {
	cout << "ReferenceMethod received:\n";
	cout << "unsigned int (reference): " << uir << '\n';
	cout << "long (reference pointer): " << *lp << '\n';
	cout << "double (reference): " << dr << endl;
}

//-------------------------------------------------------------------//

extern"C" _declspec (dllexport)
int* ArrayMethod(int count, int a[], char c[]) {
	cout << "ArrayMethod...\n";

	int i;

	cout << "Characters received are:\n";
	for (i = 0; i < count; i++) cout << c[i] << '\n';

	for (i = 0; i < count; i++) a[i] += 5;

	return a;
}

///////////////////////////////////////////////////////////////////

BOOL   WINAPI   DllMain(HINSTANCE   hinst,   DWORD   fdwReason,   LPVOID
lpvReserved) {
	switch(fdwReason) {
		case DLL_PROCESS_ATTACH:
		break;
```

```
            case DLL_THREAD_ATTACH:
            break;
            case DLL_THREAD_DETACH:
            break;
            case DLL_PROCESS_DETACH:
            break;
      }
      return TRUE;
}
```

Now, suppose we have built the C++ DLL and named it `CppSample.dll`. Our next step is to create the Z++ source file that defines the **linkage module** for dynamic linking with `CppSample.dll`. The following comments explain a few points about the Z++ source `ZppSample.zpp` listed below for linkage module.

Z++ compiler generates a warning for the following statement.

```
char* cp = "I am character pointer.";
```

Here, the compiler allocates space for `cp` and copies the Z++ **string** to it as a null-terminated C string. The compiler does not know when you will no longer need the space so it leaves the **delete** action to you. As you see, we are deleting `cp` at end of program.

That same warning is generated for `pp` in the following. That is because a literal string is a Z++ **string** object. Here, `pp` is receiving its value from the string object `stg`.

```
string stg = "I am a string.";
char* pp = stg;
```

So with regard to **string** and **`char*`** you just need to pay attention to compiler's warning and **delete** the **`char*`** when you no longer need it. This is only for passing a Z++ **string** as argument to an **external** method. When an **external** method returns a **`char*`** you simply receive the return into a Z++ **string** and the compiler deals with the conversion, and there is no warning to watch for. This is illustrated by **external** method `StringMethod()`.

Now about arrays. Here is the array example from `CppSample.cpp` listed above.

```
extern"C" _declspec (dllexport)
int* ArrayMethod(int count, int a[], char c[]);
```

The exported function `ArrayMethod()` receives two arrays and returns an array in the style of C, as a pointer. Now, here is the Z++ corresponding **external** method.

```
external int* ArrayMethod(int, int*, char*);
```

So basically for Z++ change the type of parameters from arrays to pointers. And here is the call to the **external** method, in the following program. As you can see, the arguments A and C are Z++ arrays. The compiler allows casting Z++ arrays to pointers for passing as arguments to C++ functions that expect arrays.

```
int* AR = S.ArrayMethod(count, (int*) A, (char*) C);
```

Now, you are receiving the array returned by the call in a pointer `AR`. That is not a Z++ array in the context of the linkage module. The `AR` is point to somewhere within the DLL. The following line makes an array in the context of the Z++ program, where the array `R` is declared a little earlier.

```
for (i = 0; i < count; i++) R[i] = *AR++; endfor;
```

That is all. Below is the example for Z++ linkage module. Note that we are linking with an existing C++ DLL (dynamic library), and therefore we are not using the specification **static**.

```
// ZppSample.zpp

#include<iostream.h>
using namespace ioSpace;

class Simple  "C++" persistent = "~/CppSample.dll" // dynamic linkage

      generated "~/ZppSample";

public:

      external double SimpleMethod(int, double, short);
      external ulong UnsignedMethod(uint, long, uchar);
      external int* PointerMethod(int***, char*);
      external void BooleanMethod(boolean, char**);
      external char* StringMethod(char*);
      external void ReferenceMethod(const uint&, long*&, double&);
      external int* ArrayMethod(int, int*, char*);

end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      Simple S; // Instance of linkage module

// Passing simple arguments.

      int i = 85;
      double d = 3.7;
      short s = 3;

      output << "Calling SimpleMethod(). Received : ";
      output << S.SimpleMethod(i, d, s) << '\n';
```

```
// Passing unsigned arguments.

      uint ui = 66;
      long l = 77;
      uchar uc = 22;

      output << "Calling UnsignedMethod(). Received : ";
      output << S.UnsignedMethod(ui, l, uc) << '\n';

// Pointers

      int*** ippp = &(&(&i));

// Z++ string as null-terminated C string.

      char* cp = "I am character pointer.";

// Pointers, and char*.

      output << "Calling PointerMethod(). Received : ";
      output << *S.PointerMethod(ippp, cp) << '\n';

// Boolean and char**.

      boolean bl = True;

      output << "Calling BooleanMethod() with void return.\n";
      S.BooleanMethod(bl, &cp);

// Z++ string.

      string stg = "I am a string.";
      char* pp = stg;

      output << "Calling StringMethod(). Received : ";
      string retStg = S.StringMethod(pp);
      output << retStg << '\n';

// Passing by references.

      output << "Calling ReferenceMethod() with void return.\n";
      S.ReferenceMethod(ui, &l, d);

// Passing and receiving arrays.

      const int count = 3;
      int A[count];
      A[0] = 1; A[1] = 5; A[2] = 7;

      char C[count];
      C[0] = 'A'; C[1] = 'B'; C[2] = 'C';

      int R[count];

      output << "Calling ArrayMethod()...\n";
      int* AR = S.ArrayMethod(count, (int*) A, (char*) C);
```

```
        for (i = 0; i < count; i++) R[i] = *AR++; endfor;

        output << "Values of returned array are:\n";

        for (i = 0; i < count; i++) output << R[i] << '\n'; endfor;

// Now we manually delete pointers allocated by compiler.

        delete cp;
        delete pp;

        output << "Good-bye World!\n";

end;
```

As you recall, when you compile the above program the Z++ Compiler generates a C++ source file. The **generated** statement specifies the name of the generated file as `ZppSample.cpp`. Note that the filename for generated statement should not include the file extension. The final step is then to build a DLL name `ZppSample.dll` from the C++ source `ZppSample.cpp` and copy it to the directory specified by **generated** statement.

When you run this program, the C++ DLL sends its output to the console as well. We show the outputs separately to avoid confusion.

The output of C++ DLL is as follows.

```
SimpleMethod received:
int: 85 double: 3.7 short: 3
SimpleMethod returning (double): 91.7
UnsignedMethod received:
unsigned int: 66
long: 77
unsigned char: –
UnsignedMethod returning (unsigned long): 1452
PointerMethod received:
int***: 85
char*: I am character pointer.
PointerMethod returning (int*).
BooleanMethod received boolean and it was true.
BooleanMethod also received string : I am character pointer.
StringMethod returning string : I am a string.
ReferenceMethod received:
unsigned int (reference): 66
long (reference pointer): 77
double (reference): 3.7
ArrayMethod...
Characters received are:
A
B
C
```

And the output of Z++ program is as follows.

```
Hello World!
Calling SimpleMethod(). Received : 91.7
Calling UnsignedMethod(). Received : 1452
Calling PointerMethod(). Received : 85
Calling BooleanMethod() with void return.
Calling StringMethod(). Received : I am a string.
Calling ReferenceMethod() with void return.
Calling ArrayMethod()...
Values of returned array are:
6
10
12
Good-bye World!
```

Section III.5.F. Remote Linkage and RPC

There are two forms of remote action associated with Z++ **native** modules. In one form, at the point of creation of an instance representing a module, Z47 downloads the module to the local node from a remote location. In this case the **execution** of the module takes place on the **local node**.

In the other form, Z47 requests a remote Z47 to load and run the executable of the module. In this form the **execution** of the module takes place on a **remote node** and the two Z47 Processors exchange data in **binary** format for calls to the **external** methods.

The specifications **local** and **remote** are for the **location of execution** of the module. The default is **local** execution. Consider the following line, from program `Module.zpp` in section Local Modules.

```
struct LocalModule = "~/Component.zxe"
```

If the executable `Component.zxe` were on a remote node, we would indicate the **URL** as follows.

```
struct LocalModule = "Zpp://xx.xx.xx.xx/~/Component.zxe"
```

The string of xx is the IP address of a node on which a Z++ Internet Server is listening. The tilda ~ is the path to the executable `Component.zxe`. This specification tells local Z47 to download the executable `Component.zxe` from the location indicated by the IP address, and run it locally. It is equivalent to the following, where **local** is the default.

```
struct LocalModule = "Zpp://xx.xx.xx.xx/~/Component.zxe"<local>
```

On the other hand, if we wish to have the remote Z47 load and run the executable `Component.zxe` for us, we simply replace **local** with **remote**.

```
struct LocalModule = "Zpp://xx.xx.xx.xx/~/Component.zxe"<remote>
```

As you can see, all combinations of loading/executing locally/remotely are localized to the definition of a module, specifically to the **URL**. This makes it possible to create components and the modules using those components, without having to think about where components may reside, or the location of their execution.

The conclusion is that, you can develop and test all components on a single computing device, regardless of where the components will reside, and how they will be used. All that will be needed for changes in locations of residence, as well as execution, is a localized modification to the URL, and the rest is the responsibility of the Z47 Processor, not yours.

The term **remote linkage** is used to indicate the first form of notion of remote that we have described. That is, the executable is downloaded and executed locally. On the other hand **RPC** (remote procedure call) or **RMI** (remote method invocation) indicate remote execution of a call to a global function, or a method of a class.

The notions of Z++ modules and components are more general than remote linkage and RPC or RMI. A Z++ component is a program that Z47 Processor loads as a process. For instance, a component may by multi-threaded. In that case, regardless of calls to its **entry** points, its threads will receive their time-slices and execute. Indeed, **a remote component may itself interact with other remote components, recursively**.

Instead of remote linkage and RPC, we simply say a module executes locally or remotely. A call to an **external** method simply invokes an **entry** point of a component, whether the component is a process of Z47 on the local node, or a remote node.

Section III.5.G. Linkage Module and SOA

A service may need to take advantage of the facilities of a computing device, via system calls, in providing itself to the outside world. This requires the use of a native system programming language such as C, or C++. As illustrated in this chapter, a simple DLL wrapper is all, that one needs for creating a Z++ **linkage module** in order to use the services provided in C and C++.

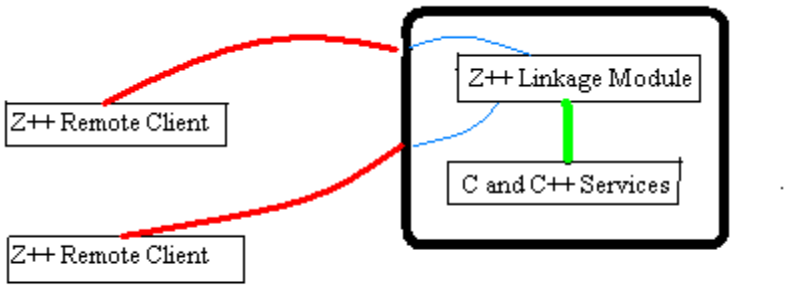
However, a linkage module, unlike native Z++ module, cannot directly communicate with a remote component written in another language. The process of building a linkage module requires that the dynamic libraries of target language to be available at execution time. This allows the passing of **pointers** and **arrays** to the calls, as well as **pass by reference**.

A native Z++ module interacts with a (remote/local) linkage module simply as a Z++ native component. On the other hand, linkage module interacts with components created in a language other than Z++. This simple design is the mechanism for Service-Oriented Architecture (SOA).

A large array of services may exist on a set of nodes (a network of hosts/servers as in **cloud computing**). The availability of such services is limited by the linguistic abstractions for the design and implementation of SOA. The combination of Z++ native modules and linkage modules provides a simple and reliable abstraction for SOA.

Each category of services can be wrapped in a DLL for loading by a Z++ linkage module. Once a linkage module is made available for a category of services, clients requesting those services can interact with this linkage module via Z++ modules from any location.

In the following diagram we are not showing the Z++ Internet Server because its operation is transparent to the idea we are describing here. The server simply needs to be up and running on the node where the services reside. The server listens for incoming requests and connects the client with the requested linkage module. The rest is between the client and the linkage module. In turn, the linkage module communicates with the C and C++ components that provide the requested service.



Service-Oriented Architecture

An example will show the simplicity of implementation of SOA in Z++. The basic idea is that the Z++ source that includes the linkage module is also a Z++ component.

For our example we will modify the Z++ program for Namespace Example. First we make couple of cosmetic changes.

1. Remove iostream because components run on remote nodes and their output is irrelevant.

2. Add **generated** specification so C++ files will not be generated. Changes we are making do no affect the libraries written in C.

The significant step is the addition of **entry** points for each of the services. At the end of the example you will find the list of added **entry** points. This allows other Z++ programs to load the executable `MathFunctions.zxe` as a native Z++ component and invoke its **entry** points, which are precisely the services provided by the C library.

The example `MathFunctions.zpp` listed below is a **typical** example of a Z++ **linkage module** that is also a Z++ **component**. Z++ **clients** receive the services written in C and C++ by loading this component.

Following the listing, we show examples of Z++ clients.

```
// MathFunctions.zpp

// ------------------------------------------------------------------ //
// Trig functions.
// ------------------------------------------------------------------ //

protected namespace trigSpace

struct TrigFunctions "C++" persistent generated = "~/TrigLib.dll"

       generated "~/TrigFunctions";

       external double ZppCos(double);
```

```
      external double ZppSin(double);
      external double ZppTan(double);

      external double ZppAcos(double);
      external double ZppAsin(double);
      external double ZppAtan(double);
end;

endspace;

// ------------------------------------------------------------------ //
// Hyperbolic functions.
// ------------------------------------------------------------------ //

protected namespace hyperSpace

struct HyperFunctions "C++" persistent generated = "~/HyperLib.dll"

      generated "~/HyperFunctions";

      external double ZppCosh(double);
      external double ZppSinh(double);
      external double ZppTanh(double);
end;

endspace;

// ------------------------------------------------------------------ //
// Logarithm and exponential functions.
// ------------------------------------------------------------------ //

protected namespace logSpace

struct LogFunctions "C++" persistent generated = "~/LogLib.dll"

      generated "~/LogFunctions";

      external double ZppExp(double);
      external double ZppLog(double);
      external double ZppLog10(double);
end;

endspace;

// ------------------------------------------------------------------ //

namespace mathSpace : trigSpace, hyperSpace, logSpace

private: // Make our work private, only export final functions

      struct MathFunctions : TrigFunctions, HyperFunctions,
                             LogFunctions
      end;

      MathFunctions MathObject;

public:
```

```
        double cos(double x) return MathObject.ZppCos(x); end;
        double sin(double x) return MathObject.ZppSin(x); end;
        double tan(double x) return MathObject.ZppTan(x); end;

        double acos(double x) return MathObject.ZppAcos(x); end;
        double asin(double x) return MathObject.ZppAsin(x); end;
        double atan(double x) return MathObject.ZppAtan(x); end;

        double cosh(double x) return MathObject.ZppCosh(x); end;
        double sinh(double x) return MathObject.ZppSinh(x); end;
        double tanh(double x) return MathObject.ZppTanh(x); end;

        double exp(double x) return MathObject.ZppExp(x); end;
        double log(double x) return MathObject.ZppLog(x); end;
        double log10(double x) return MathObject.ZppLog10(x); end;

endspace;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

using namespace mathSpace;

// ---------------------------------------------------------------- //

entry void main(void)
end;

// ---------------------------------------------------------------- //
// Trig functions.
// ---------------------------------------------------------------- //

entry double Cos(double d)
      return cos(d);
end;

entry double Sin(double d)
      return sin(d);
end;

entry double Tan(double d)
      return tan(d);
end;

entry double Acos(double d)
      return acos(d);
end;

entry double Asin(double d)
      return asin(d);
end;

entry double Atan(double d)
      return atan(d);
```

```
end;

// ------------------------------------------------------------- //
// Hyperbolic functions.
// ------------------------------------------------------------- //

entry double Cosh(double d)
      return cosh(d);
end;

entry double Sinh(double d)
      return sinh(d);
end;

entry double Tanh(double d)
      return tanh(d);
end;

// ------------------------------------------------------------- //
// Logarithm and exponential functions.
// ------------------------------------------------------------- //

entry double Exp(double d)
      return exp(d);
end;

entry double Log(double d)
      return log(d);
end;

entry double Log10(double d)
      return log10(d);
end;
```

Here is an example of a Z++ client using the services written in C.

```
//Client.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////

struct MathLib = "~/MathFunctions.zxe"

      external double Cos(double);
      external double Sin(double);
      external double Tan(double);
end;

// ------------------------------------------------------------- //

const double pi = 3.1415;

////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------- //
```

```
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      MathLib ML;

      output << ML.Cos(pi/4) << '\n';
      output << ML.Sin(pi/4) << '\n';
      output << ML.Tan(pi/4) << '\n';


      output << "Good-bye World!\n";
end;
```

Using the **namespace** mechanism we can make math library functions retain their usual familiar names, as shown below.

```
//NamespaceClient.zpp

#include<iostream.h>

// ------------------------------------------------------------------ //

namespace MathSpace

private:

      struct MathLib = "~/MathFunctions.zxe"

            external double Cos(double);
            external double Sin(double);
            external double Tan(double);
      end;

      MathLib ML;

public:

      const double pi = 3.1415;

      double cos(double d) return ML.Cos(d); end;
      double sin(double d) return ML.Sin(d); end;
      double tan(double d) return ML.Tan(d); end;

endspace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      using namespace ioSpace;
      using namespace MathSpace;
```

```
        output << "Hello World!\n";

        output << cos(pi/4) << '\n';
        output << sin(pi/4) << '\n';
        output << tan(pi/4) << '\n';

        output << "Good-bye World!\n";
end;
```

Output for both of the above clients is the following.

```
Hello World!
0.707123
0.70709
0.999954
Good-bye World!
```

Chapter III.6. Union

Type constructor **union** allows the use of multiple members taking the same space. Members of **union** in Z++ are limited to fundamental numeric types and pointers. The largest size of an instance of a union is 8 bytes, the size of a **double**.

Type constructor **union** does not support derivation. Furthermore, all members and methods are **public**. Operator overloading and conversion operators are supported. The default constructor initializes the space for members to 0. The default destructor does nothing. The default assignment and comparison operators use bit-wise operations, which is generally the intended action. As usual, user can override all of these.

Below is an example illustrating the use of **union**.

```
// Include IO stream for output

#include<iostream.h>
using namespace ioSpace;

// Define a union type

union testUnion
      int i;
      double d;

      testUnion(int);
      testUnion(double);

      operator double(void); // conversion operator
      int operator*^(int); // raise to a power
end;

// Implementation of methods for union testUnion

testUnion::testUnion(int n)
      i = n;
end;

testUnion::testUnion(double b)
      d = b;
end;

testUnion::operator double(void)
      return d;
end;

int testUnion::operator*^(int n)
      return i *^ n;
end;

// Define and enumeration for tag
```

```
enum kinds {_intKind, _doubleKind};

// Define a structure to use the union with tag

struct owner
      kinds kind;
      testUnion tu;

      owner(int);
      owner(double);

      int getIntValue(void);
      void convert(testUnion);
end;

// Implementation of methods for structure owner

owner::owner(int n) : tu(n)
      kind = _intKind;
end;

owner::owner(double b) : tu(b)
      kind = _doubleKind;
end;

int owner::getIntValue(void)
      switch(kind)
            case _intKind:
                  return tu.i;

            case _doubleKind:
                  return int(tu.d);

            else return -1;
      endswitch;
end;

void owner::convert(testUnion u)
      switch(kind)
            case _intKind:
                  tu.i = u *^ tu.i;

            case _doubleKind:
                  tu.d = u;
      endswitch;
end;

// The main program

entry void main(void)
      output << "Hello World!\n";

// ------------------------------------------------------------------ //

      int aint = 7;
      double adbl = 3.9;
```

```
        owner wint(aint);
        owner wdbl(adbl);

        output << "wint value is (7): " << wint.getIntValue() << '\n';
        output << "wdbl value is (3): " << wdbl.getIntValue() << '\n';

        testUnion t;

        t.i = 3;
        wint.convert(t);

        t.d = 47.97;
        wdbl.convert(t);

        output << "wint value is (2187): " << wint.getIntValue() << '\n';
        output << "wdbl value is (47): " << wdbl.getIntValue() << '\n';

// ------------------------------------------------------------- //

        output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
wint value is (7): 7
wdbl value is (3): 3
wint value is (2187): 2187
wdbl value is (47): 47
Good-bye World!
```

Section III.6.A. Assignment and union members

When you provide a definition for the special operators of assignment and comparison, the compiler automatically calls those operators on the members. However, when the member is union, the compiler does not call those operators.

The example of this section illustrates the reason. The default assignment for **union** copies memory. But generally **union** members are **pointers**. As seen in the example, the assignment **operator** of the structure owning the **union** member, deals with the pointers. Therefore, the compiler default is entirely unnecessary.

Keep in mind that, when you define the assignment operator, or either of the comparison operators, for a structure that has **union** members, you will need to deal with the union member. **The compiler will do nothing about union members in this case**.

```
// Union.zpp

#include<iostream.h>
using namespace ioSpace;
```

```
////////////////////////////////////////////////////////////////////////
//          This example illustrates the intended use of union
//--------------------------------------------------------------------//
// Types FirstType and SecondType will be used as members of a union.
////////////////////////////////////////////////////////////////////////

struct FirstType
      int i;

      FirstType(int);
end;

FirstType::FirstType(int n)
      i = n;
end;

//--------------------------------------------------------------------//

struct SecondType
      double d;

      SecondType(double);
end;

SecondType::SecondType(double b)
      d = b;
end;

////////////////////////////////////////////////////////////////////////

// This enumeration will serve as tagging the union

enum UnionTags {_firstType, _secondType};

//--------------------------------------------------------------------//
// UnionType will have a pointer for each type that we need.
// Union members can only be numeric types like char, int and
// double, and pointers.
// For each type in our union we define a constructor.
//--------------------------------------------------------------------//

union UnionType
      FirstType* first;
      SecondType* second;

      UnionType(FirstType*);
      UnionType(SecondType*);
end;

UnionType::UnionType(FirstType* f)
      first = f;
end;

UnionType::UnionType(SecondType* s)
      second = s;
end;
```

```
//-------------------------------------------------------------------//
// TypeManager is the type that we are going to interact with.
// It maintains a tag and an instance of union.
//-------------------------------------------------------------------//
// It is the responsibility of TypeManager to manage its union using
// its tag, as it relates to construction, destruction and assignment.
//-------------------------------------------------------------------//
// Remark. Unions can overload assignment and comparison operators,
// as well as other operators. However, unlike classes,
// the default assignment and comparison for unions is done
// point-wise (bits are copied over, or compared). That is,
// compiler does not generate default assignment and
// comparison operators for unions as it does for classes.
// This is because the compiler would not know
// the type of union member to delete, or to assign to.
//-------------------------------------------------------------------//

struct TypeManager
      UnionTags tag;
      UnionType kind;

      TypeManager(void);
      TypeManager(int);
      TypeManager(double);
      ~TypeManager(void);
      TypeManager& operator=(const TypeManager&);
end;

TypeManager::TypeManager(void)
      tag = _firstType;
      kind.first = new FirstType(0);
end;

TypeManager::TypeManager(int n)
      tag = _firstType;
      kind.first = new FirstType(n);
end;

TypeManager::TypeManager(double b)
      tag = _secondType;
      kind.second = new SecondType(b);
end;

//-------------------------------------------------------------------//
// The owning structure handles the destruction of union,
// using its tag.

TypeManager::~TypeManager(void)
      switch(tag)

            case _firstType:
                  delete kind.first;

            case _secondType:
                  delete kind.second;

      endswitch;
```

```
end;

//-----------------------------------------------------------------//
// The owning structure handles the assignment of union.
//-----------------------------------------------------------------//
// In each case, first delete the current value and create a
// new copy using the argument e.
//-----------------------------------------------------------------//

TypeManager& TypeManager::operator=(const TypeManager& e)
      if (&e == &self) return self; endif; // avoid assignment to self

      switch(tag)

            case _firstType:
                  delete kind.first;
                  kind.first = new FirstType(*e.kind.first);

            case _secondType:
                  delete kind.second;
                  kind.second = new SecondType(*e.kind.second);

      endswitch;

      return self;
end;

////////////////////////////////////////////////////////////////////
//-----------------------------------------------------------------//
////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      TypeManager ft(int(7));
      TypeManager st(47.97);

      output << "TypeManager.kind.first->i is : "
            << ft.kind.first->i << '\n';

      output << "TypeManager.kind.second->d is : "
            << st.kind.second->d << '\n';

      TypeManager aft(int(17));
      ft = aft;

      output << "After assignment, TypeManager.kind.first->i is : "
            << ft.kind.first->i << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
```

```
TypeManager.kind.first->i is : 7
TypeManager.kind.second->d is : 47.97
After assignment, TypeManager.kind.first->i is : 17
Good-bye World!
```

Chapter III.7. Bitfields

The type constructor **bitfields** facilitates the use of entities with desired number of bits and bit-wise manipulations. Members of a **bitfields** type can be from a single bit up to 32 bits long. Each member is specified with a name followed by its width (in bits). A member of **bitfields** is usually called a field.

The value of a field is independent of its position. For instance, the value of a field of 3 bits ranges between 0 and 7. The compiler produces this value in a **uint** for operations on the field or retrieval of its value. Furthermore, operands for operation with or assignment to a field are expected to be of type **uint**.

The type constructor **bitfields** receives default constructors, assignment and comparison operators. As with type constructor **class** all these defaults can be overridden. The compiler initializes all members of a **bitfields** to 0. Thus any user-defined constructor only needs to assign values to members as needed.

Type constructor **bitfields** does not support derivation (inheritance).

The following example illustrates the **bitfields** definition mechanism.

Notice the use of constructor cast `uint()` for numeric literals. The implicit type of a numeric not containing a decimal point is **long**. The compiler, for certainty, requires explicit cast to **uint**.

Overrides for the assignment and the comparison operators are unnecessary except when the engineer chooses to do something wrong. The compiler generated defaults are correct.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////

bitfields flags
      first 3;
      second 5;    // 8
      third 7;     // 15
      fourth 4;    // 19
      fifth 6;     // 25
      sixth 9;     // 34 -- crossed 32 bit boundary
      seventh 3;   // 37
      eighth 3;    // 40
      nineth 7;    // 47

      flags(void);
      flags(uint);
      flags& operator=(const flags&);
```

```
	void makeChanges(uint);

	void operator+(uint);
	uint operator+(void);

	operator uint(void); // conversion operator
end;

//-------------------------------------------------------------------//

flags::operator uint(void)
	return first;
end;

flags::flags(void)
	first = uint(5);
end;

flags::flags(uint u)
	first = u; // 5
	fourth = uint(2) * u; // 10
	sixth = uint(4) * u;  // 20
	nineth = uint(3) * u; // 15
end;

flags& flags::operator=(const flags& f)
	first = f.first;
	second = f.second;

	return self;
end;

// changer arrives with value of 3
void flags::makeChanges(uint changer)
	first |= changer;     // 5 |= 3 => 7
	second &= changer;    // 30 &= 3 => 2
	third = changer;      // 0 ==> 3
	fourth ^= changer;    // 0 ^= 3 ==> 3
	fifth = uint(2) * changer;  // 0 ==> 6
	sixth |= changer;     // 16 |= 3 ==> 19
	seventh = ~seventh;   // ~0 ==> 7
	eighth = seventh;     // 0 ==> 7
	nineth = uint(5) * changer; // 0 ==> 15
end;

void flags::operator+(uint changer)
	first -= changer;    // 4
	second *= changer;   // 6
	third /= changer;    // 1
	sixth %= changer;    // 1
	nineth += changer;   // 18
end;

uint flags::operator+(void)
	return seventh + uint(4); // 11
end;
```

```
////////////////////////////////////////////////////////////////////////

struct flagOwner
      flags f;

      flagOwner(uint);
end;

flagOwner::flagOwner(uint u) : f(u)
end;

////////////////////////////////////////////////////////////////////////

uint globalCheck(flags f)
      return f;
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
  output << "Hello World!\n";

  uint value = 11;
  flags Flag;

  output << "Flag.first at startup is (5): " << Flag.first << '\n';
  output << "Flag.fifth at startup is (0) : " << Flag.fifth << '\n';
 output << "Flag.seventh at startup is (0) : " << Flag.seventh << '\n';

  Flag.second = value;
  output << "Flag.first after assignment to second is (5) : " <<
       Flag.first << '\n';

  uint Uint;
  Uint = Flag.second;

  output << "Uint is : " << Uint << '\n';

  flags FlagTwo;
  FlagTwo = Flag;

  if (FlagTwo == Flag) output << "Bitfields are equal.\n";
  else output << "Bitfields are NOT equal.\n";
  endif;

  Uint = 55;
  Uint = FlagTwo.second;
  output << "Uint is : " << Uint << '\n';

  output << "Flag.first PRIOR to operations is : " <<
         Flag.first << '\n';

  output << "Flag.fifth PRIOR to is (0) : " << Flag.fifth << '\n';
  output << "Flag.seventh PRIOR to is (0) : " << Flag.seventh << '\n';
```

```
// Anding, oring, exoring with assignment
//-----------------------------------------------------------------//

  value = 7;
  Flag.second &= value; // 3
  output << "Flag.second after &= is (3) : " << Flag.second << '\n';
  output << "value after &= is (7) : " << value << '\n';
  output << "Flag.first after operation is (5) : " <<
        Flag.first << '\n';

  output << "Flag.fifth after operation is (0) : " <<
        Flag.fifth << '\n';

  output << "Flag.seventh after operation is (0) : " <<
        Flag.seventh << '\n';

  value = 4;
  Flag.second |= value; // 7
  output << "Flag.second after |= is (7) : " << Flag.second << '\n';
  output << "Flag.first after operation is (5) : " <<
        Flag.first << '\n';

  output << "Flag.fifth after operation is (0) : " <<
        Flag.fifth << '\n';

  output << "Flag.seventh after operation is (0) : " <<
        Flag.seventh << '\n';

  value = 25;
  Flag.second ^= value; // 30
  output << "Flag.second after ^= is (30) : " << Flag.second << '\n';
  output << "Flag.first after operation is (5) : " <<
       Flag.first << '\n';

  output << "Flag.fifth after operation is (0) : " <<
       Flag.fifth << '\n';

  output << "Flag.seventh after operation is (0) : " <<
       Flag.seventh << '\n';

////////////// Anding, oring, bit inversion withOUT assignment
//-----------------------------------------------------------------//

  output << "Flag.first is : " << Flag.first << '\n';

  ulong result;
  value = 1;
  result = Flag.first & value; // 1
  output << "Result of Flag.first with & is (1) : " << result << '\n';
  output << "Flag.first after operation is (5) : " <<
       Flag.first << '\n';

  value = 2;
  result = Flag.first | value; // 7
  output << "Result of Flag.first with | is (7) : " << result << '\n';
  output << "Flag.first after operation is (5) : " <<
       Flag.first << '\n';
```

```
  value = 3;
  result = Flag.first ^ value; // 7
  output << "Result of Flag.first with ^ is (6) : " << result << '\n';
  output << "Flag.first after operation is (5) : " <<
         Flag.first << '\n';

  result = ~Flag.first; // 2
  output << "Result of ~Flag.first is (2) : " << result << '\n';
  output << "Flag.first after operation is (5) : " <<
         Flag.first << '\n';

  output << "Flag.second after operation is (30) : " <<
         Flag.second << '\n';

// Crossing boundary
//-----------------------------------------------------------------//

  output << "Flag.fifth prior to operations is (0) : " <<
         Flag.fifth << '\n';

  output << "Flag.seventh prior to operations is (0) : " <<
         Flag.seventh << '\n';

  value = 254;
  Flag.sixth = value;
  output << "Flag.sixth is (254) : " << Flag.sixth << '\n';

// Anding, oring, exoring with assignment
//-----------------------------------------------------------------//

  value = 6;
  Flag.sixth &= value;
  output << "Flag.sixth after &= is (6) : " << Flag.sixth << '\n';
  output << "Flag.fifth after operation is (0) : " <<
          Flag.fifth << '\n';

  output << "Flag.seventh after operation is (0) : " <<
          Flag.seventh << '\n';

  value = 256;
  Flag.sixth |= value;
  output << "Flag.sixth after |= is (262) : " << Flag.sixth << '\n';
  output << "Flag.fifth after operation is (0) : " <<
          Flag.fifth << '\n';

  output << "Flag.seventh after operation is (0) : " <<
          Flag.seventh << '\n';

  value = 278;
  Flag.sixth ^= value;
  output << "Flag.sixth after ^= is (16) : " << Flag.sixth << '\n';
  output << "Flag.fifth after operation is (0) : " <<
          Flag.fifth << '\n';

  output << "Flag.seventh after operation is (0) : " <<
          Flag.seventh << '\n';
```

```
/////////////// Anding, oring, bit inversion withOUT assignment
//-------------------------------------------------------------------//

  value = 48;
  result = Flag.sixth & value;
  output << "Result of Flag.sixth with & is (16) : " << result << '\n';

  result = Flag.sixth | value;
  output << "Result of Flag.sixth with | is (48) : " << result << '\n';

  result = Flag.sixth ^ value;
  output << "Result of Flag.sixth with ^ is (32) : " << result << '\n';

  output << "Flag.sixth prior to bit inversion is (16) : " <<
          Flag.sixth << '\n';

  result = ~Flag.sixth;
  output << "Result of Flag.sixth after bit inversion is (495) : " <<
          result << '\n';

  output << "Flag.sixth after all operations is (16) : " <<
          Flag.sixth << '\n';

  output << "Flag.fifth after all operations is (0) : " <<
          Flag.fifth << '\n';

  output << "Flag.seventh after all operations is (0) : " <<
          Flag.seventh << '\n';

// Calling method
//-------------------------------------------------------------------//

  Flag.makeChanges(3);

  output << "\nAfter calling makeChanges()...\n";
  output << "Flag.first now is (7) : " << Flag.first << '\n';
  output << "Flag.second now is (2) : " << Flag.second << '\n';
  output << "Flag.third now is (3) : " << Flag.third << '\n';
  output << "Flag.fourth now is (3) : " << Flag.fourth << '\n';
  output << "Flag.fifth now is (6) : " << Flag.fifth << '\n';
  output << "Flag.sixth now is (19) : " << Flag.sixth << '\n';
  output << "Flag.seventh now is (7) : " << Flag.seventh << '\n';
  output << "Flag.eighth now is (7) : " << Flag.eighth << '\n';
  output << "Flag.nineth now is (15) : " << Flag.nineth << '\n';

  value = 3;
  Flag + 3;

  output << "\nAfter calling operator+(uint)...\n";
  output << "Flag.first now is (4) : " << Flag.first << '\n';
  output << "Flag.second now is (6) : " << Flag.second << '\n';
  output << "Flag.third now is (1) : " << Flag.third << '\n';
  output << "Flag.sixth now is (1) : " << Flag.sixth << '\n';
  output << "Flag.nineth now is (18) : " << Flag.nineth << '\n';

  output << "\nNow operator+(void)....\n";
```

```
output << "Result is (11) : " << +Flag << '\n';

output << "\nNow Comparisons....\n";

if (Flag.first < Flag.second) output << " < tested correctly.\n";
else output << " < failed the test.\n";
endif;

if (Flag.nineth > Flag.second) output << " > tested correctly.\n";
else output << " > failed the test.\n";
endif;

if (Flag.third == Flag.sixth) output << " == tested correctly.\n";
else output << " == failed the test.\n";
endif;

if (Flag.first < Flag.second < Flag.nineth) output <<
        "Sequence < tested correctly.\n";
else output << "Sequence < failed the test.\n";
endif;

output << "\nTesting ctor declarator with/without initializer...\n";

value = 5;
flags FlagThree(value);
output << "\nAfter 'FlagThree(5)'...\n";
output << "FlagThree.first now is (5) : " << FlagThree.first << '\n';
output << "FlagThree.second now is (0) : " <<
        FlagThree.second << '\n';

output << "FlagThree.third now is (0) : " <<
        FlagThree.third << '\n';

output << "FlagThree.fourth now is (10) : " <<
        FlagThree.fourth << '\n';

output << "FlagThree.fifth now is (0) : " << FlagThree.fifth << '\n';
output << "FlagThree.sixth now is (20) : " <<
        FlagThree.sixth << '\n';

output << "FlagThree.seventh now is (0) : " <<
        FlagThree.seventh << '\n';

output << "FlagThree.eighth now is (0) : " <<
        FlagThree.eighth << '\n';

output << "FlagThree.nineth now is (15) : " <<
        FlagThree.nineth << '\n';

flags FlagFour = Flag;
output << "\nAfter 'flags FlagFour = Flag'...\n";
output << "FlagFour.first now is (4) : " << FlagFour.first << '\n';
output << "FlagFour.second now is (6) : " << FlagFour.second << '\n';
output << "FlagFour.third now is (1) : " << FlagFour.third << '\n';
output << "FlagFour.fourth now is (3) : " << FlagFour.fourth << '\n';
output << "FlagFour.fifth now is (6) : " << FlagFour.fifth << '\n';
output << "FlagFour.sixth now is (1) : " << FlagFour.sixth << '\n';
```

```
  output << "FlagFour.seventh now is (7) : " <<
          FlagFour.seventh << '\n';

  output << "FlagFour.eighth now is (7) : " << FlagFour.eighth << '\n';
  output << "FlagFour.nineth now is (18) : " <<
          FlagFour.nineth << '\n';

  output << "\nTesting structure member...\n";

  flagOwner fo(value);
  output << "flagOwner.f.first now is (5) : " << fo.f.first << '\n';
  output << "flagOwner.f.second now is (0) : " << fo.f.second << '\n';
  output << "flagOwner.f.third now is (0) : " << fo.f.third << '\n';
  output << "flagOwner.f.fourth now is (10) : " << fo.f.fourth << '\n';
  output << "flagOwner.f.fifth now is (0) : " << fo.f.fifth << '\n';
  output << "flagOwner.f.sixth now is (20) : " << fo.f.sixth << '\n';
  output << "flagOwner.f.seventh now is (0) : " <<
          fo.f.seventh << '\n';

  output << "flagOwner.f.eighth now is (0) : " << fo.f.eighth << '\n';
  output << "flagOwner.f.nineth now is (15) : " << fo.f.nineth << '\n';

  value = 2;
  fo.f.first |= value;
  output << "\nflagOwner.f.first now is (7) : " << fo.f.first << '\n';

  value = Flag;
  output << "Conversion operator retured: " << value << '\n';
  output << "globalCheck() retured: " << globalCheck(Flag) << '\n';

  output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Flag.first at startup is (5): 5
Flag.fifth at startup is (0) : 0
Flag.seventh at startup is (0) : 0
Flag.first after assignment to second is (5) : 5
Uint is : 11
Bitfields are NOT equal.
Uint is : 11
Flag.first PRIOR to operations is : 5
Flag.fifth PRIOR to is (0) : 0
Flag.seventh PRIOR to is (0) : 0
Flag.second after &= is (3) : 3
value after &= is (7) : 7
Flag.first after operation is (5) : 5
Flag.fifth after operation is (0) : 0
Flag.seventh after operation is (0) : 0
Flag.second after |= is (7) : 7
Flag.first after operation is (5) : 5
Flag.fifth after operation is (0) : 0
Flag.seventh after operation is (0) : 0
Flag.second after ^= is (30) : 30
```

```
Flag.first after operation is (5) : 5
Flag.fifth after operation is (0) : 0
Flag.seventh after operation is (0) : 0
Flag.first is : 5
Result of Flag.first with & is (1) : 1
Flag.first after operation is (5) : 5
Result of Flag.first with | is (7) : 7
Flag.first after operation is (5) : 5
Result of Flag.first with ^ is (6) : 6
Flag.first after operation is (5) : 5
Result of ~Flag.first is (2) : 2
Flag.first after operation is (5) : 5
Flag.second after operation is (30) : 30
Flag.fifth prior to operations is (0) : 0
Flag.seventh prior to operations is (0) : 0
Flag.sixth is (254) : 254
Flag.sixth after &= is (6) : 6
Flag.fifth after operation is (0) : 0
Flag.seventh after operation is (0) : 0
Flag.sixth after |= is (262) : 262
Flag.fifth after operation is (0) : 0
Flag.seventh after operation is (0) : 0
Flag.sixth after ^= is (16) : 16
Flag.fifth after operation is (0) : 0
Flag.seventh after operation is (0) : 0
Result of Flag.sixth with & is (16) : 16
Result of Flag.sixth with | is (48) : 48
Result of Flag.sixth with ^ is (32) : 32
Flag.sixth prior to bit inversion is (16) : 16
Result of Flag.sixth after bit inversion is (495) : 495
Flag.sixth after all operations is (16) : 16
Flag.fifth after all operations is (0) : 0
Flag.seventh after all operations is (0) : 0

After calling makeChanges()...
Flag.first now is (7) : 7
Flag.second now is (2) : 2
Flag.third now is (3) : 3
Flag.fourth now is (3) : 3
Flag.fifth now is (6) : 6
Flag.sixth now is (19) : 19
Flag.seventh now is (7) : 7
Flag.eighth now is (7) : 7
Flag.nineth now is (15) : 15

After calling operator+(uint)...
Flag.first now is (4) : 4
Flag.second now is (6) : 6
Flag.third now is (1) : 1
Flag.sixth now is (1) : 1
Flag.nineth now is (18) : 18

Now operator+(void)....
Result is (11) : 11

Now Comparisons....
 < tested correctly.
```

```
 > tested correctly.
 == tested correctly.
Sequence < tested correctly.

Testing ctor declarator with/without initializer...

After 'FlagThree(5)'...
FlagThree.first now is (5) : 5
FlagThree.second now is (0) : 0
FlagThree.third now is (0) : 0
FlagThree.fourth now is (10) : 10
FlagThree.fifth now is (0) : 0
FlagThree.sixth now is (20) : 20
FlagThree.seventh now is (0) : 0
FlagThree.eighth now is (0) : 0
FlagThree.nineth now is (15) : 15

After 'flags FlagFour = Flag'...
FlagFour.first now is (4) : 4
FlagFour.second now is (6) : 6
FlagFour.third now is (1) : 1
FlagFour.fourth now is (3) : 3
FlagFour.fifth now is (6) : 6
FlagFour.sixth now is (1) : 1
FlagFour.seventh now is (7) : 7
FlagFour.eighth now is (7) : 7
FlagFour.nineth now is (18) : 18

Testing structure member...
flagOwner.f.first now is (5) : 5
flagOwner.f.second now is (0) : 0
flagOwner.f.third now is (0) : 0
flagOwner.f.fourth now is (10) : 10
flagOwner.f.fifth now is (0) : 0
flagOwner.f.sixth now is (20) : 20
flagOwner.f.seventh now is (0) : 0
flagOwner.f.eighth now is (0) : 0
flagOwner.f.nineth now is (15) : 15

flagOwner.f.first now is (7) : 7
Conversion operator retured: 4
globalCheck() retured: 4
Good-bye World!
```

Chapter III.8. Collection

A **collection** is a type definition mechanism that may be thought of as a generalization of enumeration, where the values can be objects of user-defined type. Ordinarily, the values associated with enumeration literals are of type **int**. However, in case of **collection** those values are instances of **struct**, **class** or **task** types.

Collection type constructor allows linear derivation (single inheritance). Methods can be specified as **public**, **protected** or **private**, with **public** being default specification. Values and the tag of a collection can be specified **protected**, and **visible**.

Methods common among classes that constitute the values of a collection can be specified as **shared collection** methods. Shared methods use commonality among classes **horizontally** in contrast to vertical sharing of inheritance mechanism. Shared methods can be specified **protected**, but not **private**.

Section III.8.A. Defining a Collection

In order to define a **collection**, we need an enumeration type and the types for its values. Let us begin by defining the following enumeration type.

```
enum basicFigures {_square, _rectangle, _triangle};
```

Next let us define three **class** types, all derived from the **class** Shape. The derivation is **not required** for defining the **collection**, though.

```
// Base class for values

class Shape

protected:

      double firstDim;
      double secondDim;

public:

      Shape(double, double);
      double area(void);
end;

Shape::Shape(double f, double s)
      firstDim = f;
      secondDim = s;
end;

double Shape::area(void)
      return firstDim * secondDim;
end;
```

Next, we derive three classes Square, Rectangle and Triangle from the **class** Shape for values of **collection**.

```
//------------------------------------------------------------------//

class Square : Shape

public:

      Square(double);
      double area(void);
      double ShowArea(void);
end;

Square::Square(double d) : Shape(d, d)
end;

double Square::area(void)
      return Shape::area();
end;

double Square::ShowArea(void)
      double myArea = Shape::area();
      output << "I am Square and my area is: " << myArea << '\n';
      return myArea;
end;

//------------------------------------------------------------------//

class Rectangle : Shape

public:

      Rectangle(double, double);
      double area(void);
      double ShowArea(void);
end;

Rectangle::Rectangle(double f, double s) : Shape(f, s)
end;

double Rectangle::area(void)
      return Shape::area();
end;

double Rectangle::ShowArea(void)
      double myArea = Shape::area();
      output << "I am Rectangle and my area is: " << myArea << '\n';
      return myArea;
end;

//------------------------------------------------------------------//

class Triangle : Shape
```

```
public:

      Triangle(double, double);
      double area(void);
      double ShowArea(void);
end;

Triangle::Triangle(double f, double s) : Shape(f, s)
end;

double Triangle::area(void)
      return Shape::area()/2;
end;

double Triangle::ShowArea(void)
      double myArea = Shape::area()/2;
      output << "I am Triangle and my area is: " << myArea << '\n';
      return myArea;
end;
```

Now we have the ingredients for defining a **collection** type. Note that the method `area()` is inherited from **class** Shape, but that the method `ShowArea()` is common among the classes with no relationship to inheritance. We will specify `ShowArea()` as **shared** method of **collection**. Explanations will follow.

We are also including the overloading of the assignment and the comparison operators. This is not needed because the compiler defaults are exactly as you see their definitions here. They are included for illustrative purposes only.

```
collection basicFiguresType<basicFigures> {
            _square<Square>,
            _rectangle<Rectangle>,
            _triangle<Triangle>

      basicFiguresType(double, double, double, double);
      basicFiguresType& operator=(const basicFiguresType&);
      boolean operator==(const basicFiguresType&) const;

shared:

      double ShowArea(void);
};
```

The name of **collection** type being defined is `basicFiguresType`, and its associated enumeration type is `basicFigures`, defined earlier. The values of collection are the classes we defined earlier. Each value is attached to a literal of the enumeration type associated with the **collection**. For instance the value at `_square` is the **class** `Square`.

Here are the definitions for the methods of **collection**.

```
basicFiguresType::basicFiguresType(double s, double l,
                                   double w, double h)
     : Square(s), Rectangle(l, w), Triangle(l, h)
```

```
end;

basicFiguresType&
basicFiguresType::operator=(const basicFiguresType& e)

      self[_square] = e[_square];
      self[_rectangle] = e[_rectangle];
      self[_triangle] = e[_triangle];
      return self;
end;

boolean basicFiguresType::operator==(const basicFiguresType& e)
      if (self[_square]!= e[_square]) return False;
      elsif (self[_rectangle]!= e[_rectangle]) return False;
      elsif (self[_triangle]!= e[_triangle]) return False;
      else return True;
      endif;
end;
```

In the next two sections we discuss **shared** methods and **collection** tag.

Section III.8.B. Collection Tag

As usual, **self** inside the body of a method refers to the instance that owns that method. The bracket function in the definition of methods of **collection** `basicFiguresType` is used on literal values of its associated enumeration. As illustrated earlier, the bracket function evaluates to the integer value of enumeration literal. When this is passed to the **collection** object it evaluates to the **class** instance that is the value of the enumeration literal. Thus, the following statement is an assignment between two instances of Triangle.

```
self[_triangle] = e[_triangle];
```

Once we reach a value of **collection** we can manipulate it in the usual way. For instance, the following will invoke the method `area()` on the instance of Square, as a value of **collection**.

```
self[_square].area();
```

An instance of **collection** internally maintains a tag. The tag is just one of the literal values of its associated enumeration. The default is the lowest value. For instance, at construction the tag for our collection example will be `_square`. It is possible to change this tag and to get its current value. Indeed, the bracket function also works on instances of **collection**. Below we are using the bracket function on **self**. However, outside of a method you will need to replace **self** with the identifier for your **collection** instance.

```
switch([self]) // bracket function used on collection

      case _square:
            output << "Area of square: " << self[_square].area();
      case _rectangle:
            output << "Area of rectangle: " << self[_rectangle].area();
      case _triangle:
            output << "Area of triangle: " << self[_triangle].area();
endswitch;
```

In order to illustrate modifying a **collection** tag, suppose we make the following declaration, and immediately we decide to set its tag to `_triangle`.

```
BasicFiguresType Figures;
[Figures] = _triangle; // change collection tag
```

That is, the bracket function yields the current value of a **collection** tag, and it can also be used to change the value of the tag.

Section III.8.C. Shared Methods

The method `ShowArea(void)` appears in **shared** section of collection**.** The keyword **shared** is used similar to **private** and **public**, but is independent of them. The shared section of a **collection** begins with the keyword **shared**, and extends to end of definition of the **collection**. Below is the list of things to observe.

- Access of the **shared** section will be whatever it was just before the **shared** keyword was visited, as to **protected** or **public**.
- Access of **shared** section can be changed in the usual way right after the keyword **shared** is visited.
- A **shared** method must be a **public** method of all the classes used for the values of the **collection**.

The last requirement is so the compiler can generate the body for a **shared** method of a **collection**. Engineer is not allowed to define the body of a **shared** method. When you call a **shared** method on a **collection** object, the method will be called on all the class objects that you have set as values of the **collection**.

Section III.8.D. Shared Method Inheritance

Shared methods of a **collection** are inherited, and cannot be re-introduced in the derived **collection**. In addition, the new class values of the extended **collection** must have the **shared** method because compiler will generate code for calling the **shared** method of a **base** on all values, including the values of the derived **collection**.

Now suppose, during derivation, you decide to introduce a new shared-method. For instance, the new **class** values of the extended collection may have another **shared** method, which is not shared by the class values in the base collection. In that case you specify the new method as **shared**, but it will only be called on the classes of the extended **collection**. This pattern continues recursively.

The notion of **shared** methods of a **collection** handles the cases where one would like to call a method on a set of classes that are not necessarily related through an inheritance hierarchy. Since the compiler generates the body of a collection's **shared** method, there will be no error in missing the call to one of the classes.

Section III.8.E. Extending Collections

In this subsection we extend the **collection** `BasicFiguresType` we defined earlier in this section. To extend a **collection**, first we need to extend its associated enumeration type. Below we are extending the enumeration type `basicFigures` to include two more values.

```
enum moreFigures : basicFigures {_parallelogram, _diamond};
```

Next, we derive two more classes from Shape for the values of our derived **collection**. Note again that class inheritance is **not** necessary for values of a **collection**.

```
//-----------------------------------------------------------------//

class Parallelogram : Shape

public:

      Parallelogram(double, double);
      double area(void);
      double ShowArea(void);
end;

Parallelogram::Parallelogram(double f, double s) : Shape(f, s)
end;

double Parallelogram::area(void)
      return Shape::area();
end;

double Parallelogram::ShowArea(void)
      double myArea = Shape::area();
      output << "I am P-gram and my area is: " << myArea << '\n';
      return myArea;
end;

//-----------------------------------------------------------------//

class Diamond : Shape

public:

      Diamond(double, double);
      double area(void);
      double ShowArea(void);
end;

Diamond::Diamond(double f, double s) : Shape(f, s)
end;

double Diamond::area(void)
      return Shape::area();
end;
```

```
double Diamond::ShowArea(void)
      double myArea = Shape::area();
      output << "I am Diamond and my area is: " << myArea << '\n';
      return myArea;
end;
```

And here is our derived (extended) **collection**.

```
collection moreFiguresType<moreFigures> : basicFiguresType {
            _parallelogram<Parallelogram>,
            _diamond<Diamond>

    moreFiguresType(double, double, double, double, double);

};

moreFiguresType::moreFiguresType(double s, double l,
                                 double w, double h, double r)
    : basicFiguresType(s, l, w, h), Parallelogram(w, r), Diamond(h, r)
end;
```

Remember that the derived **collection** `moreFiguresType` has inherited the **shared** method `ShowArea()`. In particular, note that we are not re-introducing a previously specified **shared** method. However, the class-values for the extended **collection** (Diamond and Parallelogram) must define this **shared** method.

Section III.8.F. Invoking a Shared Method

A **shared** method is invoked on **collection** object. The **collection** object will internally invoke the method on each of its (**class**) values. As mentioned earlier, the **compiler generates the body** of a **shared** method for a **collection**.

Here is an example using our earlier definitions. We create an instance of the extended **collection** `moreFiguresType`, and invoke its **shared** method.

```
moretFiguresType Figures(12, 11, 10, 8, 4);
Figures.ShowArea(); // shared method
```

The above code will print the following.

```
I am P-gram and my area is: 40
I am Diamond and my area is: 32
I am Square and my area is: 144
I am Rectangle and my area is: 110
I am Triangle and my area is: 44
```

Section III.8.G. Protecting tag and values

The **collection** tag and values are **public** by default, but can be made **protected**. In order to allow read access to the tag and values you can also specify **visible**. Below is the syntax for making values **protected** and **visible**, for **collection** `moreFiguresType` we defined earlier.

```
collection moreFiguresType<moreFigures : protected, visible>
      : basicFiguresType {
            _parallelogram<Parallelogram>,
            _diamond<Diamond>

    moreFiguresType(double, double, double, double, double);

};
```

That is, the specification **protected**, or **protected** and **visible** follows a **colon** right after the enumeration associated with the **collection** being defined. Note that the specification applies to the tag as well as to the values of **collection**. The values for this **collection** are instances of types `Parallelogram` and `Diamond`. Since we derived this **collection** from the base **collection** `basicFiguresType` the values of `basicFiguresType` remain **public** by default. That is, each **collection** must protect its own values and tag.

Section III.8.H. Collection Templates

Values for a **collection** can be instantiations of templates. In the example below we use a single **template** structure with different instantiations for values of a **collection**. However, this is only done to keep the example simple. The values for defining a **collection** can be any mixture of types and templates with any number of type parameters.

```
// Template for values of collection

template<type X, type Y> struct valueType
      X x;
      Y y;

      valueType(X, Y);
end;

template<type X, type Y> valueType::valueType(X a, Y b)
      x = a;
      y = b;
end;

// Enumeration for defining collection

enum colEnum {_one, _two, _three};
```

Below is the definition of a **collection template**. Technically, the **collection** was not defined via a template mechanism. Nonetheless, the definition is an instantiation of templates, and therefore we refer to it as **collection template**.

```
collection testCollection<colEnum> {
      _one<valueType<int, double> >,
      _two<valueType<int, int> >,
      _three<valueType<double, double> >

      testCollection(int, double);
};

testCollection::testCollection(int n, double d)
      : valueType<int, double>(n, d),
        valueType<int, int>(n, n),
        valueType<double, double>(9.7, 97.47)
end;
```

Note the space between > and > in `_one<valueType<int, double> >`. When defining a none-default constructor, simply refer to the values of the **collection** as you instantiated them, as in `valueType<int, double>(n, d)`. The compiler will replace that with its internal name for the instantiated **template** type.

Below is an example code that uses the **collection template** we just defined.

```
testCollection tc(7, 4.7); // create an instance

output << tc[_one].x << '\n';
output << tc[_one].y << '\n';
output << tc[_two].x << '\n';
output << tc[_two].y << '\n';
output << tc[_three].x << '\n';
output << tc[_three].y << '\n';
```

The output is the following.

```
7
4.7
7
7
9.7
97.47
```

Section III.8.I. Polymorphism and Casting

The example in this section illustrates polymorphism and casting with regard to single-inheritance mechanism of **collection** type constructor. Before the example we explain a few points.

Three collection types are defined: base, derivedCollection and finalCollection. The first type is base, derivedCollection is derived from base and finalCollection is derived from derivedCollection.

In `main()`, we first see a polymorphic assignment.

```
finalCollection* fcptr = new finalCollection();
base* bcptr = fcptr;
bcptr->show();
```

As you see in the output of the program, the last line invokes the `show()` of `fcptr`, which is of type `finalCollection`. This is the basic idea of polymorphism. A pointer to an object of derived type can be assigned to a pointer of the base type, but the method will be invoked on the actual object.

The difference between polymorphic assignment and casting can be seen from the output of the following lines.

```
bcptr = cast(base*, fcptr);
bcptr->show();
```

In this case the method `show()` of type base is invoked. That is because **cast** operator returns a pointer to the object, to whose type it was cast.

As in the case of **class**, casting down to a base is safe. However, casting up to a derived type needs care. The point is that, when you create an instance of a derived type such as finalCollection, **all its bases are created**. However, when you create an instance of a type that is used as base in a derivation, instances of derived types **are not created**. But the compiler cannot determine that because it is only aware of relationships among types.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////
// Enumerations for defining collections
// ------------------------------------------------------------- //

enum primes {_three = 3, _five = 5, _seven = 7, _eleven = 11};
```

```
enum morePrimes : primes {_thirteen = 13,
      _seventeen = 17,
      _nineteen = 19};

enum manyPrimes : morePrimes {_twentyThree = 23,
      _twentyNine = 29,
      _thirtyOne = 31};

////////////////////////////////////////////////////////////////////////
// Values for collection type base.
// ------------------------------------------------------------------ //

struct Three
      int i;

      Three(void);
      int show(void);
end;

Three::Three(void)
      i = 3;
end;

int Three::show(void)
      return i;
end;

// ------------------------------------------------------------------ //

struct Five
      int i;

      Five(void);
      int show(void);
end;

Five::Five(void)
      i = 5;
end;

int Five::show(void)
      return i;
end;

// ------------------------------------------------------------------ //

struct Seven
      int i;

      Seven(void);
      int show(void);
end;

Seven::Seven(void)
      i = 7;
end;
```

```
int Seven::show(void)
      return i;
end;

// ------------------------------------------------------------- //

struct Eleven
      int i;

      Eleven(void);
      int show(void);
end;

Eleven::Eleven(void)
      i = 11;
end;

int Eleven::show(void)
      return i;
end;

////////////////////////////////////////////////////////////////////////
// Values for collection type derivedCollection.
// ------------------------------------------------------------- //

struct Thirteen
      int i;

      Thirteen(void);
      int show(void);
end;

Thirteen::Thirteen(void)
      i = 13;
end;

int Thirteen::show(void)
      return i;
end;

// ------------------------------------------------------------- //

struct Seventeen
      int i;

      Seventeen(void);
      int show(void);
end;

Seventeen::Seventeen(void)
      i = 17;
end;

int Seventeen::show(void)
      return i;
end;
```

```
// ------------------------------------------------------------------ //

struct Nineteen
      int i;

      Nineteen(void);
      int show(void);
end;

Nineteen::Nineteen(void)
      i = 19;
end;

int Nineteen::show(void)
      return i;
end;

////////////////////////////////////////////////////////////////////////
// Values for collection type finalCollection.
// ------------------------------------------------------------------ //

struct TwentyThree
      int i;

      TwentyThree(void);
      int show(void);
end;

TwentyThree::TwentyThree(void)
      i = 23;
end;

int TwentyThree::show(void)
      return i;
end;

// ------------------------------------------------------------------ //

struct TwentyNine
      int i;

      TwentyNine(void);
      int show(void);
end;

TwentyNine::TwentyNine(void)
      i = 29;
end;

int TwentyNine::show(void)
      return i;
end;

// ------------------------------------------------------------------ //

struct ThirtyOne
      int i;
```

```
      ThirtyOne(void);
      int show(void);
end;

ThirtyOne::ThirtyOne(void)
      i = 31;
end;

int ThirtyOne::show(void)
      return i;
end;

////////////////////////////////////////////////////////////////////////
// Defining collection types.
// ------------------------------------------------------------------ //

collection base<primes> {
      _three<Three>,
      _five<Five>,
      _seven<Seven>,
      _eleven<Eleven>

      void show(void);
};

void base::show(void)
      output << "\nPrinting values of base...\n";
      output << self[_three].show() << '\n';
      output << self[_five].show() << '\n';
      output << self[_seven].show() << '\n';
      output << self[_eleven].show() << '\n';
end;

// ------------------------------------------------------------------ //

collection derivedCollection<morePrimes> : base {
      _thirteen<Thirteen>,
      _seventeen<Seventeen>,
      _nineteen<Nineteen>

      void show(void);
};

void derivedCollection::show(void)
      output << "\nPrinting values of derivedCollection...\n";
      output << self[_thirteen].show() << '\n';
      output << self[_seventeen].show() << '\n';
      output << self[_nineteen].show() << '\n';
end;

// ------------------------------------------------------------------ //

collection finalCollection<manyPrimes> : derivedCollection {
      _twentyThree<TwentyThree>,
      _twentyNine<TwentyNine>,
      _thirtyOne<ThirtyOne>
```

```
      void show(void);
};

void finalCollection::show(void)
      output << "\nPrinting values of finalCollection...\n";
      output << self[_twentyThree].show() << '\n';
      output << self[_twentyNine].show() << '\n';
      output << self[_thirtyOne].show() << '\n';
end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      finalCollection* fcptr = new finalCollection();

// Polymorphic assignment

      base* bcptr = fcptr;
      bcptr->show();

// Casting down to a base collection

      bcptr = cast(base*, fcptr);
      bcptr->show();

// casting up to a derived collection

      derivedCollection* dcptr = cast(derivedCollection*, bcptr);
      dcptr->show();

      delete fcptr;

      output << "\nGood-bye World!\n";

end;
```

The output of the example is as follows.

```
Hello World!

Printing values of finalCollection...
23
29
31

Printing values of base...
3
5
7
11
```

```
Printing values of derivedCollection...
13
17
19

Good-bye World!
```

Chapter III.9. Typedef

The **new operator** does not accept a type followed by * for its argument. For instance, consider the following, which causes **syntax error**.

```
int** p = new int*; // syntax error
```

Instead, you need to use **typedef** as follows.

```
typedef IntPointer int*;
int** p = new IntPointer;
```

However, the above approach does not initialize the pointer p. Consider the following instead, which prints 7.

```
typedef IntPointer int*;
int argument = 7;
int** p = new IntPointer(&argument);
output << **p << '\n';
```

The operands of **typedef** are, first the **new type name**, followed by the type. This is in **opposite** direction to the C syntax. Typedef has many uses shown in various examples in this book. For instance, see Degenerate Arrays and Template Typedef.

The new type name does not define a new type. That means, in the following example, `firstIntPointer` and `secondIntPointer` are the same type. The compiler will not complain when making assignments, as shown below.

```
typedef firstIntPointer int*;
typedef secondIntPointer int*;
firstIntPointer first;
secondIntPointer second;
first = second; // no error
```

Typedef introduces a new type name and for that reason cannot be **included** inside a **function definition**.

Typedef is not the mechanism for defining new function types. The mechanism for that purpose is **type**, illustrated in Type and Function Pointer.

Chapter III.10. Type and Function pointer

A function is an object with a specific type. Therefore, like other types we should be able to define a pointer to a function type. A **function pointer** is simply an instance of a function (pointer) type.

The mechanism for defining a function type is **type** and not **typedef**. Examples will illustrate some of the reasons for this choice. But the main difference is that **type** creates a **new type**, while **typedef** is simply a renaming of an **existing type**. More specifically, instances of two function types **cannot** be assigned to one another. The two function pointers must be instance of the same function type. Obviously this type checking applies to passing arguments to function calls.

In the following example we define the **function type** `MyFunType`. Notice the simplicity of the syntax. It is the word **type** followed by a **prototype**. The name of the prototype becomes the name of **function type**. Notice that for historical reasons we are referring to an instance of a function type as **function pointer**.

In the example, you also see a function `globalFun(int i, long g)`. Later in main() we define the following.

```
MyFunType MyFun = globalFun;
```

The above line is a regular declaration, with initial value. Here is the example.

```
// SampleOne.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

// Define a function type
type int MyFunType(int, long);

// Parameter is function type
void receiver(MyFunType f)
      f(2, 3);
end;

// Function to be pointed to by MyFunType.
int globalFun(int i, long g)
      output << "In globalFun, got : " << i << ' ' << g << '\n';
      return i;
end;

void globalFunPointerTester(int n)
      output << "In globalFunPointerTester(), got : " << n << '\n';
end;
```

```
////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// declare and initialize an instance of MyFunType

      MyFunType MyFun = globalFun;

      output << "Calling receiver ...\n";

// pass the instance of MyFunType to a function

      receiver(MyFun);

// execute MyFun in place, and pass its return object to the call

      globalFunPointerTester(MyFun(2,3));

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Calling receiver ...
In globalFun, got : 2 3
In globalFun, got : 2 3
In globalFunPointerTester(), got : 2
Good-bye World!
```

Function pointers can also be **members of structures**, as illustrated by next example.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

type void MyFunType(int, long);

void globalFun(int i, long g)
      output << "In globalFun, got : " << i << ' ' << g << '\n';
end;

// ------------------------------------------------------------------ //

struct memberFunPointer
      int n;
      long l;
```

```
      MyFunType FPtr;

      memberFunPointer(int, long, MyFunType);
      void testerMethod(void);
end;

memberFunPointer::memberFunPointer(int t, long g, MyFunType fp)
      n = t;
      l = g;
      FPtr = fp;
end;

void memberFunPointer::testerMethod(void)
      FPtr(n, l);
end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      MyFunType MyFun = globalFun;

      memberFunPointer MFP(6, 7, MyFun);

      MFP.testerMethod();

      MFP.FPtr(13, 55);

      output << "Good-bye World!\n";
end;
```

The output of the example is as follows.

```
Hello World!
In globalFun, got : 6 7
In globalFun, got : 13 55
Good-bye World!
```

For further examples illustrating the use of **type** see also examples in chapters on Throws Specification, Inter-thread Communication and Registering Hear Signals.

Parameter default initializers are not allowed for function pointer types, as explained in that appendix.

Chapter III.11. Pointer

Compiler constructs pointer types implicitly. In the following declaration, the second * is not really needed. However, in order to avoid confusion, Z++ keeps the tradition started with C.

```
int* p, *q;
```

In the above statement, compiler constructs a new type **int***, unless is has done so in a previous declaration. It then creates the instance p. Therefore, it can simply create the instance q of type **int***. Thus, the * preceding q is redundant. However, as we shall see in next section about arrays, the redundancy is not all that bad. The **typedef** mechanism helps with a more readable form of pointer declaration.

```
typedef IntPointer int*;
IntPointer p, q;
```

The term type modifier is not used in Z++. A pointer type is an actual type, not a modified form of a type. In Z++ the only literal value for a pointer object is 0. The address operator, as shown below, represents all other values for pointer objects.

```
int IntObject = 5;
int* p = &IntObject;
```

It is better to view pointer objects as variables that range over a set of objects, rather than a pointer being **another name** for the **same object**. For instance, when traversing a linked list `node = node->next`, in a loop, the view that node is just another name for the same object does not seem appropriate.

In order to avoid copying when passing large objects to function calls, use pass by reference. A pointer parameter should be used when in fact the pointer as an object needs to be manipulated by the called function.

Section III.11.A. Pointer Casting

The only C-style casting allowed in Z++ is pointer casting. This is mainly with regard to **void*** or casting when passing arguments to C++ dynamic libraries. The C-style of casting for pointers will be referred to as **pointer casting**.

Pointer casting does nothing beyond telling the compiler to temporarily consider an object to be of another type. This is **unsafe** and should be used with care. On the other hand, the **cast operator** looks for conversions, and provides a means of moving up and down in an inheritance tree, as illustrated in Inheritance and Casting.

Section III.11.B. Pointer De-referencing

Operators have a predefined precedence order. We can always use parentheses to enforce our intended order of precedence. An interesting case of precedence is between increment (decrement) **operator** and dereferencing a pointer.

The expression *p++ where p is pointer may seem to mean, dereference pointer and then increment the result (value of *p). Actually, both operators are applied to the pointer p. So, first the pointer is derenferenced and its value made available to the expression containing it. Then, the operator ++ is applied to the pointer p, not *p.

What you need to remember is that de-referencing returns what a pointer is pointing to, **without changing the pointer**. Well then, the operator ++ is applied to the pointer. For instance consider the expression (*p)++. Here the operator ++ is applied to what is inside the parentheses, which is the value *p, and not the pointer p.

In Z++, it is not possible to rewrite the expression (*p)++, as ++*p. Actually ++*p causes syntax error. The reason for this is because the expression ++*p is ambiguous. First dereferencing the pointer and using its value, and after that incrementing the pointer would be same as *p++. On the other hand, incrementing the value of *p (as done in C), violates the view we described with regard to *p++. That is, in this case we are saying the operator ++ is applied to the value of *p. Well, we should say the same to *p++.

The expression *++p can only mean increment the pointer and then dereference it. That causes no ambiguity.

Below is a sample for this section.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////
//----------------------------------------------------------------//
```

```
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      char charArray[6];
      charArray[0] = 'H';
      charArray[1] = 'e';
      charArray[2] = 'l';
      charArray[3] = 'l';
      charArray[4] = 'o';
      charArray[5] = 0;

      char* p = (char*) charArray;

      output << "ch will be same for both outputs...\n";

      char ch = *++p;
      output << ch << '\n';
      ch = *p;
      output << ch << '\n';

      output << "ch will be different for each output...\n";

      ch = *p++;
      output << ch << '\n';
      ch = *p;
      output << ch << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
ch will be same for both outputs...
e
e
ch will be different for each output...
e
l
Good-bye World!
```

Chapter III.12. Array

Z++ support for arrays is quite extensive. Regular (static) as well as dynamic arrays are supported. You can also create both kinds of arrays via the **new operator**. Numeric arrays can be used in expression just like simple numeric objects. Conversions and overloading of operators extends to arrays.

An array of any type is itself a type. For instance, the type of `intArray` in the following declaration is `int[3][5]`.

```
int intArray[3][5];
```

The type for a dynamic array is called **degenerate**. Consider the following declaration.

```
double doubleArray[dimOne][dimTwo];
```

Here, `dimOne` and `dimTwo` are two integers whose values are unknown at compile time. The type of `doubleArray` is `double[][]`. That is, the type is a **two-dimensional degenerate array of doubles**.

Section III.12.A. Static and Dynamic Arrays

An array is **static** when, the sizes of its dimensions are known to the compiler. Whether the array was created by the **new operator** or not, is not a factor. On the other hand, when the compiler cannot determine the sizes of dimensions, the array is **dynamic**.

The following example illustrates declaring and initializing static and dynamic arrays of fundamental types.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

typedef intArrayType int[3][5];
typedef doubleArrayType double[][];

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      int intArray[3][5](231); // static array
```

```
        output << "intArray[2][2] is: " << intArray[2][2] << '\n';

        int dimOne = 3, dimTwo = 5;
        double doubleArray[dimOne][dimTwo](1.2); // dynamic array

        output << "doubleArray[2][2] is: " << doubleArray[2][2] << '\n';

        intArrayType* iap = new int[3][5](379); // static array

        output << "(*iap)[2][2] is: " << (*iap)[2][2] << '\n';

// Dynamic array

        doubleArrayType* dap = new double[dimOne][dimTwo](3.9);

        output << "(*dap)[2][2] is: " << (*dap)[2][2] << '\n';

        delete iap;
        delete dap;

        output << "Good-bye World!\n";
end;
```

The output of the example is as follows.

```
Hello World!
intArray[2][2] is: 231
doubleArray[2][2] is: 1.2
(*iap)[2][2] is: 379
(*dap)[2][2] is: 3.9
Good-bye World!
```

Section III.12.B. Degenerate Arrays

In the example of preceding section, we had the following **typedef**.

```
typedef doubleArrayType double[][];
```

We used this typedef in the following declaration.

```
doubleArrayType* dap = new double[dimOne][dimTwo](3.9);
```

In general, when we need to declare dynamic arrays in complex ways, in particular as members of **class** type where the dimensions are unknown, we use degenerate arrays. At elaboration we supply the dimensions, like dimOne and dimTwo. A degenerate array does not have sizes for its dimensions, and cannot be constructed. It merely represents a type for a dynamic array, which will receive the sizes for its dimensions at construction time.

Section III.12.C. Sizes of dimensions of Dynamic arrays

The sizes of dimensions of a dynamic array are not known at compile time. However, for iteration statements, such as for-loop, we need this information.

At run time, the size operator can tell us what the size of each dimension of a dynamic array is. The following example illustrates the use of the **size operator**. Dimension numbering starts with 0. That is, first dimension is number 0, second dimension is number 1, and so on.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      int dimOne = 2, dimTwo = 3;

      int intArray[dimOne][dimTwo](776);
      intArray++;

      for (int first = 0; first < size(intArray, 0); first++)
            for (int second = 0; second < size(intArray, 1); second++)
                  intArray[first][second] += (first + second);
                  output << intArray[first][second] << '\n';
            endfor;
      endfor;

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
777
778
779
778
779
780
Good-bye World!
```

Section III.12.D. Array support for fundamental types

Static arrays of fundamental types can be used in expressions just like objects of fundamental types. Dynamic arrays are supported to the extent that is feasible. The following sample includes a few examples. For further information refer to the operators of your interest, like Assignment Operators, Arithmetic Operators, etc.

Array operations expect compatible arrays.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

// Static arrays

      int intArray[3][5](60);

      double doubleArray[3][5] = intArray;

      doubleArray += 44.78;

      output << "doubleArray[0][0] is: " << doubleArray[0][0] << '\n';
      output << "doubleArray[1][2] is: " << doubleArray[1][2] << '\n';
      output << "doubleArray[2][3] is: " << doubleArray[2][3] << '\n';

      intArray *= doubleArray;

      output << "intArray[0][0] is: " << intArray[0][0] << '\n';
      output << "intArray[1][2] is: " << intArray[1][2] << '\n';
      output << "intArray[2][3] is: " << intArray[2][3] << '\n';

      double resultArray[3][5] = intArray - doubleArray;

      output << "resultArray[0][0] is: " << resultArray[0][0] << '\n';
      output << "resultArray[1][2] is: " << resultArray[1][2] << '\n';
      output << "resultArray[2][3] is: " << resultArray[2][3] << '\n';

      intArray &= 240;

      output << "intArray[0][0] is: " << intArray[0][0] << '\n';
      output << "intArray[1][2] is: " << intArray[1][2] << '\n';
      output << "intArray[2][3] is: " << intArray[2][3] << '\n';

// Dynamic arrays

      int dimOne = 3, dimTwo = 5;
```

```
      string stringArray[dimOne][dimTwo]("My name is: ");

      stringArray += "Tarzan.";

      output << stringArray[0][0] << '\n';
      output << stringArray[1][2] << '\n';
      output << stringArray[2][3] << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
doubleArray[0][0] is: 104.78
doubleArray[1][2] is: 104.78
doubleArray[2][3] is: 104.78
intArray[0][0] is: 6240
intArray[1][2] is: 6240
intArray[2][3] is: 6240
resultArray[0][0] is: 6136
resultArray[1][2] is: 6136
resultArray[2][3] is: 6136
intArray[0][0] is: 96
intArray[1][2] is: 96
intArray[2][3] is: 96
My name is: Tarzan.
My name is: Tarzan.
My name is: Tarzan.
Good-bye World!
```

Section III.12.E. Arrays and Overloading operators

Structural types, class, task, collection, union etc. can overload operators as methods. The overloading of operators extends to arrays of structural types. In general, all methods of a structural type can also be called on static and dynamic arrays of those types. This means that, when a method is invoked on an array, it is actually called on every cell of the array.

One thing to keep in mind is that, the objects returned from calling each cell of the array are not usable. It is no feasible to create an array for all returned objects. For one thing, methods may have void return, but there is no void array. When the return is by reference returning an array of reference is probably not useful, but will take time to construct.

With the above argument in mind, an **array call** always returns a **reference** to the array on which the call was made. By an **array call** we mean invoking a method of the cell on the array object.

For examples please see Calls on Array Objects, and Arrays of Tasks.

Section III.12.F. Array Conversions

This topic is illustrated in the chapter on **conversions** section Array Conversions. The basic idea is as follows. A conversion operator defined for a structural type is a method. Therefore, the compiler applies the conversion to an array of such structural type.

Unlike array calls, discussed in previous section, conversions generally result in an entire converted array.

Section III.12.G. Pointer Syntax for Arrays

In keeping up with C-tradition, Z++ provides a generalized pointer syntactic sugar for accessing array cell. The simplest form, same as C, is via casting. Consider the following examples, casting a one-dimensional static array.

```
// Casting a static array

      char cArray[5];

      cArray[0] = 'A';
      cArray[1] = 'B';
      cArray[2] = 'C';
      cArray[3] = 'D';
      cArray[4] = 'E';

      char* pArray = (char*) cArray;

      char c = pArray[2];
      output << c << '\n';
```

```
output << pArray[4] << '\n';
```

This prints the letters C and E. Compiler interprets the expression `pArray[4]` as equivalent of `(*pArray)[4]`. That is, it will reach the array by dereferencing the pointer, and then move to the cell of index 4.

Casting one-dimensional dynamic arrays works the same way. Here is an example.

```
// Casting a dynamic array

      int mm = 4;
      int iArray[mm];

      for (int i = 0; i < mm; i++)
            iArray[i] = 2 * i + 7;
      endfor;

      int* pdArray = (int*) iArray;
      output << pdArray[2] << '\n';
```

This prints 11. However, casting to base type does not work correctly for arrays of higher dimensions. The type of pointer must match the type of array. Consider the following example, for both cases of static and dynamic arrays.

```
// Two dimensional static and dynamic arrays

typedef DoubleStaticArray double[2][3];
typedef DoubleDynamicArray double[][];

DoubleStaticArray* dsa = new DoubleStaticArray(4.7);
output << dsa[1][1] << '\n';

int m = 3, n = 4;
DoubleDynamicArray* dda = new double[m][n](9.7);
output << dda[2][3] << '\n';
```

The output is 4.7 and 9.7. For instance, the expression `dda[2][3]` is the equivalent of the expression `(*dda)[2][3]`.

The pointer syntax works as expected for arrays of structures. Here is the full example.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

//-----------------------------------------------------------------//

struct base
      double d;

      base(double);
```

```
end;

base::base(double b)
	d = b;
end;

//---------------------------------------------------------------------//

struct test : base
	int i;

	test(int, double);
	int operator+(double);
end;

test::test(int n, double b) : base(b)
	i = n;
end;

int test::operator+(double b)
	base::d = b;
	return i;
end;

//---------------------------------------------------------------------//

typedef DoubleStaticArray double[2][3];
typedef DoubleDynamicArray double[][];

//---------------------------------------------------------------------//

typedef StructStaticArray test[2][3];
typedef StructDynamicArray test[][];

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
	output << "Hello World!\n";

//---------------------------------------------------------------------//

	int dim1 = 3, dim2 = 4;
	StructDynamicArray* sdap = new test[dim1][dim2](7, 4.7);
	output << sdap[2][3]::base::d << '\n';

	StructStaticArray* ssap = new StructStaticArray(47, 97.47);
	output << ssap[1][2]::base::d << '\n';

//---------------------------------------------------------------------//

	char cArray[5];

	cArray[0] = 'A';
	cArray[1] = 'B';
	cArray[2] = 'C';
```

```
	cArray[3] = 'D';
	cArray[4] = 'E';

	char* pArray = (char*) cArray; // casting static array

	char c = pArray[2];
	output << c << '\n';

	output << pArray[4] << '\n';

	int mm = 4;
	int iArray[mm];

	for (int i = 0; i < mm; i++)
		iArray[i] = 2 * i + 7;
	endfor;

	int* pdArray = (int*) iArray; // casting dynamic array
	output << pdArray[2] << '\n';

//-----------------------------------------------------------------------//

	DoubleStaticArray* dsa = new DoubleStaticArray(4.7);
	output << dsa[1][1] + 3.2 << '\n';
	output << dsa[1][1] << '\n';

	int m = 3, n = 4;
	DoubleDynamicArray* dda = new double[m][n](9.7);
	output << dda[2][3] + 3.2 << '\n';
	output << dda[2][3] << '\n';

//-----------------------------------------------------------------------//

	output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
4.7
97.47
C
E
11
7.9
4.7
12.9
9.7Good-bye World!
```

Chapter III.13. Frame and Canvas

Chapter IV. Abstract Data Types (Templates)

Template is the name of a linguistic mechanism for defining Abstract Data Types. We may also refer to **template** mechanism as **Parameterized Type**, which emphasizes the fact that the instantiation of a **template** results in a **type**.

The parameters of a **template** can only be instantiated with **previously defined types**. It is possible to provide **specifications** for the **types** that can **instantiate** a **template** parameter.

In a project with multiple source files, you can separate the definition of a **template** from its implementation the same way you do for an ordinary **class** type. The include files only need the definition of the template, just like **class** types.

All mechanisms available for a **class**, such as **invariant**, **constraint**, **friend** and **visible** are also available for templates.

See Z++ Template Library (ZTL) for standard abstract data types.

Chapter IV.1. Template Definition and Instantiation

A **template** definition uses the syntax **template**<**type** parameter>, as illustrated in the following example. A simple **template** is defined, and instantiated.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

//-----------------------------------------------------------------//

template<type X, type Y> class DoubleParameterTemplate
      X first<visible>;
      Y second<visible>;

public:

      DoubleParameterTemplate(X, Y);
      void incFirst(void);
      Y addToSecond(Y);
end;

template<type X, type Y>
DoubleParameterTemplate::DoubleParameterTemplate(X f, Y s)
      first = f;
      second = s;
end;

template<type X, type Y>
```

```
void DoubleParameterTemplate::incFirst(void)
      first++;
end;

template<type X, type Y>
Y DoubleParameterTemplate::addToSecond(Y s)
      return second += s;
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Instantiate template, and create an instance

      DoubleParameterTemplate<int, double> dpt(7, 9.7);

// Call some methods

      dpt.incFirst();
      dpt.addToSecond(4.7);

// Show values

      output << dpt.first << '\n';
      output << dpt.second << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
8
14.4
Good-bye World!
```

Chapter IV.2. Template Parameter Specification (Pattern)

Instantiators for **template** parameters must be previously defined types, which includes previously instantiated **template** types. However, there are times that one defines an abstract data type for a specialized purpose. In the latter case, the category of types to be used as instantiators may require certain characteristics.

Template **pattern** is a linguistic mechanism for specifying the category of types that can be used to instantiate a **template** parameter. The **pattern** specification has two forms. In its simple form, one lists all acceptable types, as shown below.

```
pattern template<T> SimplePattern
      T : int, double;
end;
```

The **pattern** `SimplePattern` only allows **int** and **double** as instantiators.

Another form of **pattern** specification is the listing of required **public** methods from the instantiator type. Below is an example. The **question** mark for name of a method is **wild** and matches any method with same **signature**, in the instantiator type.

```
pattern template<T> ElaboratePattern
      operator T(void); // conversion operator
      T operator+=(T); // overloaded operator
      void ?(T, int); // signature for a method
end;
```

When defining a **template** type, each type parameter can use its own **pattern** specification or none, as shown below for three **template** parameters X, Y and Z. In particular, **template** parameter Y does not have a **pattern** specification, and can be instantiated with any type.

```
template<type X : SimplePattern, type Y , type Z : ElaboratePattern>
```

The example of this section uses both forms of **template pattern** specifications, in a nested **template** instantiation. An instantiation of the **template class** `StringType` is used as an **instantiator** for the outer **template class** `StringOperations` satisfying the conditions required by the **pattern** `OperationPattern`.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//
// Template pattern specifying acceptable types.

pattern template<T> StringPattern
      T : string;
end;

//-------------------------------------------------------------------//
// An instantiation of this template will satisfy the
// conditions of next template pattern: OperationPattern.
// At the same time, it only accepts string type as
// an instantiator for its template parameter.

template<type T : StringPattern> class StringType
      T elem;
public:

      StringType(void);
      StringType(T);
```

```
      operator T(void);
      T operator+=(T);

end;

template<type T> StringType::StringType(void) : elem()
end;

template<type T> StringType::StringType(T e) : elem(e)
end;

template<type T> StringType::operator T(void)
      return elem;
end;

template<type T> T StringType::operator+=(T e)
      elem += e;
      return elem;
end;

//-------------------------------------------------------------------//
// Template pattern specifying methods

pattern template<T> OperationPattern
      operator T(void);
      T operator+=(T);
end;

//-------------------------------------------------------------------//
// This template can only take instantiators that satisfy
// the conditions of pattern: OperationPattern.

template<type T : OperationPattern> class StringOperations
      T elem;
public:
      StringOperations(void);
      StringOperations(T);
      T combine(T);
end;

template<type T> StringOperations::StringOperations(void)
end;

template<type T> StringOperations::StringOperations(T e) : elem(e)
end;

template<type T> T StringOperations::combine(T e)
      elem += e;
      return elem;
end;

/////////////////////////////////////////////////////////////////////
//-----------------------------------------------------------------//
/////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";
```

```
// A nested instantiation using template patterns

      StringOperations<StringType<string> >
           stgOps(StringType<string>("Hi there."));

// A simple use of template pattern

      StringType<string> stgType("Initialized");

// Next line outputs the string "Hi there.Initialized".

      output << stgOps.combine(stgType) << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Hi there.Initialized
Good-bye World!
```

Chapter IV.3. Template Typedef

A **template typedef** requires the same **template** parameter specification as when defining a **template** type. Below is a line from the example that follows.

```
template<type U, type V> typedef MyBase Base<U, V>[];
```

The name of **typedef** is `MyBase` and the type it defines is `Base<U, V>[]`.

Template **typedef** is useful in the same ways that regular **typedef** is. In the following example we use **template typedef** for defining a **degenerate template** for a dynamic array.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------- //
// A template type

template<type U, type V> struct Base
      U u;
      V v;

      Base(U, V);
      void show(void);
end;
```

```
template<type U, type V>  Base::Base(U a, V b)
      u = a;
      v = b;
end;

template<type U, type V>  void Base::show(void)
      output << "u is : " << u << '\n';
      output << "v is : " << v << '\n';
end;

// ------------------------------------------------------------------ //
// One-dimensional degenerate template array.

template<type U, type V> typedef MyBase Base<U, V>[];

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      int m = 7;
      int n = 3;

// Instantiating a dynamic array of templates.

      Base<int, double> B[n](m, 3.7);
      B[1].show();

// MBP is a degenerate pointer, using typedef MyBase.

      MyBase<int, double>* MBP = new Base<int, double>[n](m, 9.7);
      (*MBP)[1].show();

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
u is : 7
v is : 3.7
u is : 7
v is : 9.7
Good-bye World!
```

Chapter IV.4. Task as Template Instantiator

An elaborate example of **task template** is provided in Task Template. The example of this section illustrates the use of **task** as an instantiator of a **template** type. A **task** type

retains its characteristics as an instantiator. There is nothing in particular to keep in mind when using **tasks** as instantiators of **template** parameters.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //
// Task for instantiation template parameter

task MultiplicationTask
public:
      int Multiply(int, int);
end;

int MultiplicationTask::Multiply(int a, int b)
      return a*b;
end;

// ------------------------------------------------------------------ //
// Task template

template <type T> task AdditionTask
public:
      int Add(T&, T&);
end;

template <type T> int AdditionTask::Add(T& a, T& b)
      return a.Multiply(6, 5) + b.Multiply(10, 11);
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      MultiplicationTask a;

// Instantiate task template with a task type

      AdditionTask<MultiplicationTask> AddInt;

      output << AddInt.Add(a, a) << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
140
Good-bye World!
```

Chapter IV.5. Template Namespace

Template **namespace** follows the same rules as discussed in Namespace. There is only one difference when separating the definition of a **namespace** from its implementation. The implementation section, as shown in Template Namespace Implementation, needs qualification with **template**. In this chapter we make a complete example.

The purpose of separating the implementation of a **namespace** from its definition is to provide the definition in a header file, and the implementation in another file. Otherwise, when using a single file we can include the definition and the implementation in the definition section of a **namespace**. However, for purposes of illustration we shall include the definition and the implementation in a single example.

```
// Example.zpp

#include<iostream.h>

// ------------------------------------------------------------------ //
// namespace definition

namespace simpleNamespace

template <type T> struct simpleTemplate

      T i;

      simpleTemplate(T);
      operator T (void);
end;

endspace;

// ------------------------------------------------------------------ //
// namespace implementation

implementation<template> simpleNamespace

template<type T> simpleTemplate::simpleTemplate(T k)
      i = k;
end;

template <type T> simpleTemplate::operator T (void)
      return i;
end;

endspace<template>;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////
```

```
entry void main(void)

      using namespace ioSpace;

      output << "Hello World!\n";

      simpleNamespace::simpleTemplate<int> sti(47);

      int k = 3;

      k += sti; // uses conversion operator of sti

      output << k << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
50
Good-bye World!
```

Chapter V. Namespace

It is desirable to choose a name for a type or an object that suggests its intended use. In a large program clash of names is inevitable. The notion of **namespace** organizes the infinite (global) space into subspaces. Each of these subspaces is called a namespaces, containing certain categories of types and possibly some instances of those types.

Namespace is a packaging concept for libraries. For instance, several vendors can provide the Z++ Template Library (ZTL), each using their own namespace. An engineer should only have to modify the name of namespace he or she is using for a successful rebuild. **The difference, if any, should only be visible in the performance of the program**.

The Z++ **namespace** construct is syntactically similar to the **class** type constructor. However, a **namespace** is not a type and therefore no instances of a **namespace** can be created. The Z++ approach to the notion of **namespace** facilitates the packaging of large libraries, as well as the systematic use of such libraries in large programs.

In an object-oriented program, a large number of objects will interact. Declaring a long list of such objects in the single main entry of a C++ program may only look inelegant. In contrast, Z++ programs allow multiple **entry** points, and **multiple entry points may be called before the program terminates**. Thus, any one of the **entry** points may need the services of a set of global objects, and we do not want these objects to be initialized each time an **entry** point is called. In this respect, **Z++ namespaces provide same level of localization for global objects as classes do for their members and methods**.

Chapter V.1. Namespace Sections

Namespaces have **private**, **protected** and **public** sections with same semantics as **class**. The **private** section is for internal use by the **namespace**. The **protected** section can be inherited in derivations, but otherwise is **private**. Only the **public** section of a **namespace** is accessible to users of the **namespace**. Thus, **the public section is what a namespace exports to the outside world**.

The default section for a **namespace** is **public**, similar to **struct** rather than **class**. In the following example the **namespace** `ExportSpace` introduces a type, and it also exports an instance of this type.

```
namespace ExportSpace

      class SpaceClass
            string name;
      public:
            SpaceClass(string);
            string get(void);
      end;
```

```
        SpaceClass scInstance; // export an instance

endspace;
```

We can provide implementations for the methods of `SpaceClass` in the above definition of the **namespace**, or we can do so in the **implementation** of the **namespace**, as illustrated in the next section.

Chapter V.2. Namespace Implementation

The definition of a **namespace** and its implementation can be **separated**. This allows including a **namespace** in multiple files of a project that are separately compiled without complications. Furthermore, it has the same readability advantages as separating the implementation of methods of a **class** from the definition of the **class** itself.

Since the notion of **namespace** is related to packaging libraries, the separation of its definition from its implementation has programmatic applications. The implementation of a **namespace** can be distributed in **binary** for linkage. Users need only include the definition of a **namespace** in their programs.

Below is an implementation of the example **namespace** presented in previous section. The syntax is straightforward. The keyword **implementation** is followed with the name of the **namespace**. The same keyword **endspace**, which closes the definition of a **namespace**, also closes the implementation.

```
implementation ExportSpace

        SpaceClass::SpaceClass(string s)
                name = s;
        end;

        string SpaceClass::get(void)
                return name;
        end;

endspace;
```

Chapter V.3. Namespace Derivation

Namespace derivation is **multiple-inheritance**. A **namespace** can only derive from other **namespaces**. The default for derivation is **public**. **Private and public derivations have the same semantics as for classes**.

Chapter V.4. Protected Namespace

A large library may end up in name clashes for its own internal use. In that case, the library may need to use the **namespace** mechanism purely for **internal matters**. This means that the library needs to introduce a set of namespaces without allowing its users to access the contents of these namespaces.

To solve the above problem, a **namespace** can be specified **protected**, as illustrated below.

```
protected namespace Name-of-namespace
// body of namespace
endspace;
```

The semantics is that, a **protected namespace** can only be used in a derivation. In particular, users cannot directly access the contents of a **protected namespace**. Since derivation could be **private**, the contents of a **protected namespace** could remain hidden from the users of the derived **namespace**.

Thus, a large library can define several **protected** namespaces in order to avoid its own internal name clashes, without affecting the users of the library.

Chapter V.5. Scope of a Namespace

A **namespace** exports it **public** contents. A user accesses the exported contents of a **namespace** as follows.

```
using namespace Name-of-namespace;
```

This makes the **public** section of the **namespace** available. It is possible to **end** the availability of a **namespace** at any point in a program, as shown below.

```
endusing namespace Name-of-namespace;
```

Public **namespace** items can also be accessed selectively, as shown below.

```
Name-of-namespace::namespaceItem
```

The above phrase can appear in any valid expression or statement. Furthermore, nested namespaces (bases of derived namespaces) can be reached recursively.

```
Namespace_1::namespace_2::…::namespace_n::Item
```

Chapter V.6. Namespace Example

In following example we define same class multiSpace in three different namespaces, as well as the global space, with minor differences. In main() we can see that different versions of the class multiSpace are kept separate by namespaces.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////
```

```
// The class multiSpace is defined in two namespaces,
// a derived namespace and in global space.
// A member is made visible for each version of same type, and
// default constructors initialize instances differently.
// ------------------------------------------------------------- //
// In main() we reach instances of these types correctly.
///////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////
// In global version of multiSpace, member s is visible.

class multiSpace
      short s<visible>;
      int n;
      double d;
public:
      multiSpace(void);
      multiSpace(short, int, double);
      short gets(void);
      int getn(void);
      double getd(void);
end;

// ------------------------------------------------------------- //

multiSpace::multiSpace(void)
      s = 0;
      n = 0;
      d = 0;
end;

multiSpace::multiSpace(short t, int i, double b)
      s = t;
      n = i;
      d = b;
end;

short multiSpace::gets(void)
      return s;
end;

int multiSpace::getn(void)
      return n;
end;

double multiSpace::getd(void)
      return d;
end;

///////////////////////////////////////////////////////////////////
// In namespace firstSpace, member n of multiSpace is visible.

namespace firstSpace
class multiSpace
      short s;
      int n<visible>;
      double d;
```

```
public:
      multiSpace(void);
      multiSpace(short, int, double);
      short gets(void);
      int getn(void);
      double getd(void);
end;

// ------------------------------------------------------------- //

multiSpace::multiSpace(void)
      s = 5;
      n = 55;
      d = 5.5;
end;

multiSpace::multiSpace(short t, int i, double b)
      s = t;
      n = i;
      d = b;
end;

short multiSpace::gets(void)
      return s;
end;

int multiSpace::getn(void)
      return n;
end;

double multiSpace::getd(void)
      return d;
end;
endspace;

////////////////////////////////////////////////////////////////////
// In namespace secondSpace, member d of multiSpace is visible.

namespace secondSpace
class multiSpace
      short s;
      int n;
      double d<visible>;
public:
      multiSpace(void);
      multiSpace(short, int, double);
      short gets(void);
      int getn(void);
      double getd(void);
end;

// ------------------------------------------------------------- //

multiSpace::multiSpace(void)
      s = 5;
      n = 55;
      d = 5.5;
```

```
end;

multiSpace::multiSpace(short t, int i, double b)
      s = t;
      n = i;
      d = b;
end;

short multiSpace::gets(void)
      return s;
end;

int multiSpace::getn(void)
      return n;
end;

double multiSpace::getd(void)
      return d;
end;
endspace;

////////////////////////////////////////////////////////////////////////
// In namespace largeSpace, member d of multiSpace is visible.

namespace largeSpace : firstSpace, secondSpace
class multiSpace
      short s;
      int n;
      double d<visible>;
public:
      multiSpace(void);
      multiSpace(short, int, double);
      short gets(void);
      int getn(void);
      double getd(void);
end;

// ------------------------------------------------------------------ //

multiSpace::multiSpace(void)
      s = 8;
      n = 88;
      d = 8.9;
end;

multiSpace::multiSpace(short t, int i, double b)
      s = t;
      n = i;
      d = b;
end;

short multiSpace::gets(void)
      return s;
end;

int multiSpace::getn(void)
      return n;
```

```
end;

double multiSpace::getd(void)
      return d;
end;
endspace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
  output << "Hello World!\n";

// Global Space

  ::multiSpace globalMS(7, 47, 97.47);

  output << "In global space s is visible : " << globalMS.s << '\n';
  output << "globalMS.n is : " << globalMS.getn() << '\n';
  output << "globalMS.d is : " << globalMS.getd() << '\n';

// firstSpace

  firstSpace::multiSpace firstMS(3, 37, 73.37);

  output << "firstMS.s is : " << firstMS.gets() << '\n';
  output << "In firstSpace n is visible : " << firstMS.n << '\n';
  output << "firstMS.d is : " << firstMS.getd() << '\n';

// secondSpace

  secondSpace::multiSpace secondMS(5, 55, 15.51);

  output << "secondMS.s is : " << secondMS.gets() << '\n';
  output << "secondMS.n is : " << secondMS.getn() << '\n';
  output << "In secondSpace d is visible : " << secondMS.d << '\n';

// largeSpace::firstSpace

  largeSpace::firstSpace::multiSpace largeFirstMS(13, 33, 3.3);

  output << "largeSpace::firstMS.s is : " <<
    largeFirstMS.gets() << '\n';

  output << "In largeSpace::firstSpace n is visible : " <<
    largeFirstMS.n << '\n';

  output << "largeSpace::firstMS.d is : " <<
    largeFirstMS.getd() << '\n';

// largeSpace

  largeSpace::multiSpace largeMS;

  output << "largeMS.s is : " << largeMS.gets() << '\n';
  output << "largeMS.n is : " << largeMS.getn() << '\n';
```

```
  output << "In largeSpace d is visible : " << largeMS.d << '\n';

  output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
In global space s is visible : 7
globalMS.n is : 47
globalMS.d is : 97.47
firstMS.s is : 3
In firstSpace n is visible : 37
firstMS.d is : 73.37
secondMS.s is : 5
secondMS.n is : 55
In secondSpace d is visible : 15.51
largeSpace::firstMS.s is : 13
In largeSpace::firstSpace n is visible : 33
largeSpace::firstMS.d is : 3.3
largeMS.s is : 8
largeMS.n is : 88
In largeSpace d is visible : 8.9
Good-bye World!
```

Chapter V.7. Template Namespace Implementation

When you decide to **separate** the definition of a **namespace** including **template** definitions from the implementation of that **namespace**, the implementation needs **template** specification, as shown below.

```
implementation<template> Name-of-Namespace
// implementation statements
endspace<template>;
```

Note that the definition of **namespace** does not need this specification, nor you have to separate the implementation from the definition. The entire work can be done within the definition of a **namespace**.

Chapter VI. The notion of Signal

Signaling can be abstracted as a relation. The elements in the domain of the relation are called signals, and those in the range are called handlers. Each signal in the domain is associated with one or more handlers in the range. It is also possible for multiple signals to be associated with the same handler.

A realization of signaling involves at least two entities. The entities agree upon the sign that represents a signal, and the action to be taken as a response for handling the signal. In programming, the delivery of signals is performed by the operating system, to which we refer as notification. Generally, programmer specifies a block of code as a handler for a signal. However, a signal that is predefined within an operating system comes with a default handler that a programmer may override. Thus, in general the programmer defines the relation that associates a set of signals to their handlers.

The source of a signal could be a software-related user request, such as shutdown. In this case the operating system broadcasts the signal by notifying all its processes to terminate. A user can also generate a hardware-related signal, such as a mouse-click. These signals are delivered to threads or processes that have the focus.

A form of synchronous signaling is known as exception. When an exception is raised against a process, its normal execution is interrupted and the handler for the exception is invoked instead. The delivery of an exception signal is confined to the process in which it is raised.

In programming, signaling is also used as a means of communication among threads and processes. Z47 Processor can also deliver **distributed signals**. Furthermore, beyond an indication of the occurrence of an event, Z++ signals can also carry along **data** to the recipients of signals.

In the following chapters we discuss and illustrate the signaling mechanism of Z++.

Chapter VI.1. Preliminary Definitions

A signal is generally used to convey a pre-determined piece of information in a specific context. Signaling is a form of communication usually with the intent that the recipient of the signal takes an agreed upon action. A signal may be sent to one or more recipients.

In software, senders and recipients of signals are processes, threads and the operating system. For our purposes, developing Z++ applications as opposed to system programs, the operating system is opaque. Below, we will use the term agent to mean a thread or a process of Z47 Processor.

Remark. Z47 Processor has two scopes. Local scope is the set of all processes under a single Z47. Distributed scope is the union of scopes of a set of cooperating Z47 Processors. The following definitions are relative to a Z47 local scope. The more general form of distributed signaling will be introduced when discussing Tell-Hear signals.

With regard to programming, we leave the notion of signal as undefined. The action of sending a signal is also referred to as generating that signal. The recipient of a signal is said to **catch** the signal. When Z47 delivers a signal to an agent we say that the signal has **arrived**. An agent can catch a signal only after it arrives. The term **handler** can be defined as the block of code executed in response to catching a signal.

Remark. The UNIX signaling mechanism is a form of exception handling. An agent registers a handler with operating system for a particular signal. When the signal is generated, the execution of the agent is interrupted and the handler is invoked. In our context an agent generates a signal, and at a later time some agent eventually responds to the signal. The process is asynchronous, and the execution of the agent that generates a signal is not interrupted.

The **extent** of a signal is the set of agents that can catch the signal. The following are the categories of extent of a signal.

Process-bounded extent. The extent of a signal is the set of all threads of the process that generated the signal.

Node-bounded extent. The extent of a signal is all agents. Depending on type of agent, this extent is further divided into following subcategories.

Thread-node-bounded extent. In this case, a signal generated by a process can be caught by, any thread of any process.

Process-node-bounded extent. In this case, a signal generated by one process can be caught by, any process.

The **propagation** of a signal is the number of agents that can catch the signal. A signal is called **singular** when this number is exactly one. Otherwise the signal is called **entire**.

A singular signal is considered handled as soon as an agent catches it. On the other hand, an entire signal can be caught by any number of agents. In the latter case, all agents that wish to catch an entire signal must **register** their desire.

Chapter VI.2. Generating and Catching signals

Now we illustrate Z++ statements related to signaling. The primitive notion of signal is a built-in object of Z++. That is, **signal** is a reserved keyword of Z++ language.

We will discuss the various types of signals shortly. For now, suppose My_Signal is a signal. An agent that wishes to generate this signal uses the following statement.

```
signal <- My_Signal;
```

The operator `<-` tells the object **`signal`** to generate the signal.

An agent that wishes to catch this signal uses the following expression.

```
signal ? My_Signal
```

This is a **boolean** expression that evaluates to true or false. If the signal has arrived the expression evaluates to true, otherwise it evaluates to false.

Chapter VI.3. Process-bounded Singular signals

All types of Z++ signals are defined in the header file exception.h and use the namespace exceptionSpace.

The simplest kind of Z++ signals, are process-bounded singular signals. Only threads of the process that generates this kind of signal can catch it. Furthermore, once the signal is caught it is removed from queue so that no other thread will be able to catch it.

The enumeration type name of this category of signals is signalEventType, and for clarity, each literal of this type is expected to have the prefix _SIGNAL. One defines new signals of this type by extending the enumeration type signalEventType as follows.

```
enum MyServiceSignals : signalEventType {
      _SIGNAL_TerminateServer,
      _SIGNAL_ClientCompleted,
      _SIGNAL_ServerInitializationComplete,
      _SIGNAL_ServerTerminationComplete
};
```

A thread usually catches these signals in a loop. For instance, the following loop will terminate after the signal `_SIGNAL_ServerTerminationComplete` arrives.

```
do
enddo(signal ? _SIGNAL_ServerTerminationComplete);
```

Inside a loop, the thread can use the following statement to break out at the arrival of a signal.

```
if (signal ? _SIGNAL_TerminateServer) break; endif;
```

Chapter VI.4. Node-bounded Singular signals

Node-bounded singular signals can be caught by, any thread of any process (within the scope of a single Z47 Processor). Once caught these signals are removed from queue so that no other thread can catch them again.

The enumeration type defining this category of signals is universalEventType. For clarity, the expected prefix for literals of this type is _UNIVERSAL_SIGNAL.

Except for a wider extent, these signals are used the same way as process-bounded singular signals, as illustrated in previous section.

Chapter VI.5. Process-bounded Entire signals

These signals do not cross the boundary of a process. When a process generates this kind of signal, all threads of the same process that have registered to catch the signal will eventually catch it. A **thread** could be global or a **task** thread. We illustrate the registration for each kind of thread in the following subsections.

The enumeration type name for this category of signals is threadEntireEventType. For clarity the literals of this type should be prefixed with _THREAD_ENTIRE.

New signals of this kind are defined by extending the type threadEntireEventType, as illustrated below.

```
enum MyEntireSignals : threadEntireEventType {
      _THREAD_ENTIRE_Signal_1,
      _THREAD_ENTIRE_Signal_2,
      _THREAD_ENTIRE_Signal_3
};
```

We will use the above definition in following sections.

Section VI.5.A. Registration for Global Threads

In this subsection we illustrate how a global **thread** registers to catch an entire signal. We are going to use the entire signals defined earlier in this chapter. Below is an example of a global thread, registering two of those signals.

```
void MyGlobalThread(void)<thread>
accepts(_THREAD_ENTIRE_Signal_2, _THREAD_ENTIRE_Signal_3)
      // Body of function
end;
```

The statement **accepts**(…) tells the compiler to register the listed signals for the global thread MyGlobalThread. That means, whenever either of the two listed signals are generated by, any **thread** of the process, the global thread MyGlobalThread will be notified of the arrival of those signals, **regardless** of how many other threads may have registered to catch them.

In order for the thread to catch the signals it has registered, it will have to use the expression discussed in earlier sections as follows.

```
signal ? _THREAD_ENTIRE_Signal_2
signal ? _THREAD_ENTIRE_Signal_3
```

The simple mechanism of pairing off each signal with a handler is not general enough to cover all uses of signals. For instance, a thread may need to handle the arrival of multiple

signals in a specific order without regard to the order in which they might have been generated. The body of `MyGlobalThread` may look like following.

```
// Wait for arrival of _THREAD_ENTIRE_Signal_2
do enddo(signal ? _THREAD_ENTIRE_Signal_2);

// Handle _THREAD_ENTIRE_Signal_2

// Wait for arrival of _THREAD_ENTIRE_Signal_3
do enddo(signal ? _THREAD_ENTIRE_Signal_3);

// Handle _THREAD_ENTIRE_Signal_3
```

The above pattern will handle `_THREAD_ENTIRE_Signal_2` before handling the arrival of signal `_THREAD_ENTIRE_Signal_3` whether or not `_THREAD_ENTIRE_Signal_2` was in fact generated before `_THREAD_ENTIRE_Signal_3`.

On the other hand, at times we need to sense both signals simultaneously and act upon the signal that has arrived. For instance, one signal could tell the thread to terminate, and the other may require certain handling. This scenario is illustrated below. The simple mechanism of pairing off each signal with its handler requires more complex coding.

```
do
      if (signal ? _THREAD_ENTIRE_Signal_2) break;
      elsif (signal ? _THREAD_ENTIRE_Signal_3)
            // Handle the signal
      endif;
enddo;
```

Section VI.5.B. Registration for Task Threads

A **task** thread registers an entire signal it wishes to handle as part of definition of **task** type, as illustrated below.

```
task MyTask

      accepts(_THREAD_ENTIRE_Signal_1);

public:
end;
```

For further illustration see Section VI.5.D below and Task Accept Signals.

Section VI.5.C. Task Handlers

A **task** type must define a handler for each signal that it intends to catch. In case of entire signals, the definition of **task** type will include **accept** statements. This helps the compiler to identify the signals that lack a handler and trap such cases as error.

We repeat the definition of previous section and add a handler to it.

```
task MyTask

      accepts(_THREAD_ENTIRE_Signal_1);

      void MyHandler(void)<_THREAD_ENTIRE_Signal_1>;

public:
end;

void MyTask::MyHandler(void)
      // Body of method
      // Does not include (signal ? _THREAD_ENTIRE_Signal_1)
end;
```

Note that the body of a **task** handler does not include the expression for catching the signal attached to the handler. Instead, the signal is attached to a handler in its definition as a prototype inside the definition of **task** type. Then, at runtime, when the signal arrives the handler is automatically invoked (on **task** object). As noted earlier the handling of a signal is asynchronous. That is, some thread generates a signal, and at a later time the **task** object is notified of the arrival of the signal, at which time the **task** object invokes the handler attached to the arrived signal.

Remark. A **task** handler cannot be invoked directly.

Remark. Task handlers can be attached to singular signals, process-bounded or node-bounded, that do not need registration.

For further illustration see the next section and Task Signal Handlers.

Section VI.5.D. Example for Entire Signals

The example of this section illustrates the use of **accepts**(…) for registering entire signals. This example must run on Z47L. The **Z++ Internet Server** allows bringing up the Server for database, autonomous agent etc. **and** it also allows bringing up the Z47L Server in listening mode. Z47L is for loading multiple Z++ applications on a single Z47 processor, and for **delivering entire signals**, as the name listening mode indicates.

```
// EntireSignals.zpp

// ------------------------------------------------------------------ //
// This program must be executed on Z47L, in listening monde.
// It registers entire signals via accepts(...) specification.
// ------------------------------------------------------------------ //
// Multiple tasks and global threads register to catch entire signals.
// ------------------------------------------------------------------ //

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
// These are simple events: singular, process-bounded. They can be
// caught by any thread in the process that generated them, only once.
// ------------------------------------------------------------------ //

enum MyOwnSignals : signalEventType {
      _SIGNAL_Thread_started_1,
      _SIGNAL_Thread_started_2,
      _SIGNAL_Thread_started_3,
      _SIGNAL_Thread_started_4,
      _SIGNAL_Thread_ended_1,
      _SIGNAL_Thread_ended_2,
      _SIGNAL_Thread_ended_3,
      _SIGNAL_Thread_ended_4,
      _SIGNAL_Task_Handler_One_1,
      _SIGNAL_Task_Handler_One_2,
      _SIGNAL_Task_Handler_Two_1,
      _SIGNAL_Task_Handler_Two_2
};

////////////////////////////////////////////////////////////////////////
// These are process-bounded entire signals. They require registeration
// for catching. All threads within same process that generates such a
// signal will eventually catch that signal.
// ------------------------------------------------------------------ //
// Z47L (listening mode) is needed for these signals.
```

```
// ------------------------------------------------------------------ //

enum MyEntireSignals : threadEntireEventType {
      _THREAD_ENTIRE_Signal_1,
      _THREAD_ENTIRE_Signal_2,
      _THREAD_ENTIRE_Signal_3,
      _THREAD_ENTIRE_Signal_4,
      _THREAD_ENTIRE_Signal_5,
      _THREAD_ENTIRE_Signal_6,
      _THREAD_ENTIRE_Signal_7,
      _THREAD_ENTIRE_Signal_8,
      _THREAD_ENTIRE_Signal_9
};

//////////////////////////////////////////////////////////////////////
// tasks One and Two use accepts(...) specification to register entire
// signals that they are willing to catch. Both tasks register same
// signals. When the signals are generated, both tasks will catch them.
// ------------------------------------------------------------------ //
// Each signal needs a corresponding handler. When signal is generated
// Z47L will deliver the signal to the task object. The task object
// after the delivery of signal, invokes its corresponding handler.
// ------------------------------------------------------------------ //
// Each handler generates a simple signal to tell the main() that
// it was invoked. This is so, the main() waits until all tasks
// complete their processing before terminating the program.
//////////////////////////////////////////////////////////////////////

task One

      accepts(_THREAD_ENTIRE_Signal_1, _THREAD_ENTIRE_Signal_5);

      void FirstHandlerOne(void)<_THREAD_ENTIRE_Signal_1>;
      void SecondHandlerOne(void)<_THREAD_ENTIRE_Signal_5>;

public:
end;

void One::FirstHandlerOne(void)
      output << "Inside FirstHandlerOne(void).";
      output << " Sending signal _SIGNAL_Task_Handler_One_1.\n";

      signal <- _SIGNAL_Task_Handler_One_1;
end;

void One::SecondHandlerOne(void)
      output << "Inside SecondHandlerOne(void).";
      output << " Sending signal _SIGNAL_Task_Handler_One_2.\n";

      signal <- _SIGNAL_Task_Handler_One_2;
end;

//////////////////////////////////////////////////////////////////////

task Two

      accepts(_THREAD_ENTIRE_Signal_1, _THREAD_ENTIRE_Signal_5);
```

```
      void FirstHandlerTwo(void)<_THREAD_ENTIRE_Signal_1>;
      void SecondHandlerTwo(void)<_THREAD_ENTIRE_Signal_5>;

public:
end;

void Two::FirstHandlerTwo(void)
      output << "Inside FirstHandlerTwo(void).";
      output << " Sending signal _SIGNAL_Task_Handler_Two_1.\n";

      signal <- _SIGNAL_Task_Handler_Two_1;
end;

void Two::SecondHandlerTwo(void)
      output << "Inside SecondHandlerTwo(void).";
      output << "Sending signal _SIGNAL_Task_Handler_Two_2.\n";

      signal <- _SIGNAL_Task_Handler_Two_2;
end;

////////////////////////////////////////////////////////////////////////
// There are four global threads that register to catch entire signals.
// _THREAD_ENTIRE_Signal_1 is shared by tasks,
// and two of the global threads.
// _THREAD_ENTIRE_Signal_5 is shared by all task and global threads.
// ------------------------------------------------------------------ //
// Each global thread informs the main() that it has started.
// This is so the thread registers its accepts(...) list
// before main() generates those signals.
// ------------------------------------------------------------------ //
// Each thread also generates a simple signal to inform main()
// that it has ended. This makes main() wait until all
// threads complete their processing before terminating the program.
////////////////////////////////////////////////////////////////////////

void FirstGlobalThread(void)<thread>
      accepts(_THREAD_ENTIRE_Signal_1,
            _THREAD_ENTIRE_Signal_2,
            _THREAD_ENTIRE_Signal_5)

      signal <- _SIGNAL_Thread_started_1; // tell main we started

      output << "FirstGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_1.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

      output << "FirstGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_2.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_2);

      output << "FirstGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_5.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_5);
```

```
      output << "FirstGlobalThread(): ending the thread.\n";
      signal <- _SIGNAL_Thread_ended_1;
end;

////////////////////////////////////////////////////////////////////////

void SecondGlobalThread(void)<thread>
      accepts(_THREAD_ENTIRE_Signal_3,
            _THREAD_ENTIRE_Signal_4,
            _THREAD_ENTIRE_Signal_5)

      signal <- _SIGNAL_Thread_started_2;

      output << "SecondGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_3.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_3);

      output << "SecondGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_4.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_4);

      output << "SecondGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_5.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_5);

      output << "SecondGlobalThread(): ending the thread.\n";
      signal <- _SIGNAL_Thread_ended_2;
end;

////////////////////////////////////////////////////////////////////////

void ThirdGlobalThread(void)<thread>
      accepts(_THREAD_ENTIRE_Signal_1,
            _THREAD_ENTIRE_Signal_2,
            _THREAD_ENTIRE_Signal_5)

      signal <- _SIGNAL_Thread_started_3;

      output << "ThirdGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_5.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_5);

      output << "ThirdGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_1.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

      output << "ThirdGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_2.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_2);
```

```
      output << "ThirdGlobalThread(): ending the thread.\n";
      signal <- _SIGNAL_Thread_ended_3;
end;

////////////////////////////////////////////////////////////////////////

void FourthGlobalThread(void)<thread>
      accepts(_THREAD_ENTIRE_Signal_3,
            _THREAD_ENTIRE_Signal_4,
            _THREAD_ENTIRE_Signal_5)

      signal <- _SIGNAL_Thread_started_4;

      output << "FourthGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_3.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_3);

      output << "FourthGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_5.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_5);

      output << "FourthGlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_4.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_4);

      output << "FourthGlobalThread(): ending the thread.\n";
      signal <- _SIGNAL_Thread_ended_4;
end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Main: Hello World!\n";

// ---------------------------------------------------------------- //
// Create task objects. Task constructors will register the signals
// ---------------------------------------------------------------- //

      output << "Main: starting tasks One and Two.\n";
      One N;
      Two T;

// ---------------------------------------------------------------- //
// Now, start each global thread and wait until the thread sends the
// signal that it has started. The creation of thread includes the
// registeration of signals in its accepts(...) list.
// ---------------------------------------------------------------- //

      output << "Main: starting FirstGlobalThread().\n";
      FirstGlobalThread();
      output << "Main: waiting for _SIGNAL_Thread_started_1.\n";
```

```
      do enddo(signal ? _SIGNAL_Thread_started_1);

      output << "Main: starting SecondGlobalThread().\n";
      SecondGlobalThread();
      output << "Main: waiting for _SIGNAL_Thread_started_2.\n";
      do enddo(signal ? _SIGNAL_Thread_started_2);

      output << "Main: starting ThirdGlobalThread().\n";
      ThirdGlobalThread();
      output << "Main: waiting for _SIGNAL_Thread_started_3.\n";
      do enddo(signal ? _SIGNAL_Thread_started_3);

      output << "Main: starting FourthGlobalThread().\n";
      FourthGlobalThread();
      output << "Main: waiting for _SIGNAL_Thread_started_4.\n";
      do enddo(signal ? _SIGNAL_Thread_started_4);

// ------------------------------------------------------------------ //
// Now generate all five entire signals. In particular the
// entire signal _THREAD_ENTIRE_Signal_5 will be caught by the
// two task threads and four global threads.
// ------------------------------------------------------------------ //

      output << "Main: sending signal _THREAD_ENTIRE_Signal_1.\n";
      signal <- _THREAD_ENTIRE_Signal_1;

      output << "Main: sending signal _THREAD_ENTIRE_Signal_2.\n";
      signal <- _THREAD_ENTIRE_Signal_2;

      output << "Main: sending signal _THREAD_ENTIRE_Signal_3.\n";
      signal <- _THREAD_ENTIRE_Signal_3;

      output << "Main: sending signal _THREAD_ENTIRE_Signal_4.\n";
      signal <- _THREAD_ENTIRE_Signal_4;

      output << "Main: sending signal _THREAD_ENTIRE_Signal_5.\n";
      signal <- _THREAD_ENTIRE_Signal_5;

// ------------------------------------------------------------------ //
// Wait until the four global threads signal their completion.
// ------------------------------------------------------------------ //

      output << "Main: waiting for _SIGNAL_Thread_ended_1.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_1);

      output << "Main: waiting for _SIGNAL_Thread_ended_2.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_2);

      output << "Main: waiting for _SIGNAL_Thread_ended_3.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_3);

      output << "Main: waiting for _SIGNAL_Thread_ended_4.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_4);

// ------------------------------------------------------------------ //
// Wait until all task handlers signal their completion.
// ------------------------------------------------------------------ //
```

```
      output << "Main: waiting for _SIGNAL_Task_Handler_One_1.\n";
      do enddo(signal ? _SIGNAL_Task_Handler_One_1);

      output << "Main: waiting for _SIGNAL_Task_Handler_One_2.\n";
      do enddo(signal ? _SIGNAL_Task_Handler_One_2);

      output << "Main: waiting for _SIGNAL_Task_Handler_Two_1.\n";
      do enddo(signal ? _SIGNAL_Task_Handler_Two_1);

      output << "Main: waiting for _SIGNAL_Task_Handler_Two_2.\n";
      do enddo(signal ? _SIGNAL_Task_Handler_Two_2);

// ------------------------------------------------------------------ //

      output << "Main: Good-bye World!\n"; // We are all done

end;
```

The output of this example is as follows.

Note. The blank lines are not part of the output. They have been added because word processor has broken the lines into two, making it hard to understand output.

```
Main: Hello World!
Main: starting tasks One and Two.
Main: starting FirstGlobalThread().
FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.
Main: waiting for _SIGNAL_Thread_started_1.
Main: starting SecondGlobalThread().
SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_3.
Main: waiting for _SIGNAL_Thread_started_2.
Main: starting ThirdGlobalThread().
ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.
Main: waiting for _SIGNAL_Thread_started_3.
Main: starting FourthGlobalThread().
FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_3.
Main: waiting for _SIGNAL_Thread_started_4.
Main: sending signal _THREAD_ENTIRE_Signal_1.
Main: sending signal _THREAD_ENTIRE_Signal_2.

Inside FirstHandlerOne(void). Sending signal
_SIGNAL_Task_Handler_One_1.

Inside FirstHandlerTwo(void). Sending signal
_SIGNAL_Task_Handler_Two_1.

FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_2.
Main: sending signal _THREAD_ENTIRE_Signal_3.
Main: sending signal _THREAD_ENTIRE_Signal_4.
FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.
SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_4.
FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.
Main: sending signal _THREAD_ENTIRE_Signal_5.
```

```
Main: waiting for _SIGNAL_Thread_ended_1.

Inside SecondHandlerOne(void). Sending signal
_SIGNAL_Task_Handler_One_2.

Inside SecondHandlerTwo(void). Sending signal
_SIGNAL_Task_Handler_Two_2.

FirstGlobalThread(): ending the thread.
SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.
ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.
FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_4.
Main: waiting for _SIGNAL_Thread_ended_2.
SecondGlobalThread(): ending the thread.
ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_2.
FourthGlobalThread(): ending the thread.
Main: waiting for _SIGNAL_Thread_ended_3.
ThirdGlobalThread(): ending the thread.
Main: waiting for _SIGNAL_Thread_ended_4.
Main: waiting for _SIGNAL_Task_Handler_One_1.
Main: waiting for _SIGNAL_Task_Handler_One_2.
Main: waiting for _SIGNAL_Task_Handler_Two_1.
Main: waiting for _SIGNAL_Task_Handler_Two_2.
Main: Good-bye World!
```

Chapter VI.6. Node-bounded Entire signals

Node-bounded entire signals propagate among **processes**, as opposed to threads. That is, when a process generates a signal of this kind, all processes that have registered to be notified, including the process that generated the signal, will in fact catch this signal.

Once a process catches a node-bounded entire signal, only one thread of that process will be able to serve the signal. That is, only one single thread in each process will be able to catch this kind of signal because as soon as a **thread** catches the signal, it will be removed from the queue of the **process**.

The enumeration type name for this category of signals is processEntireEventType. For clarity the literals of this type should be prefixed with _PROCESS_ENTIRE.

New signals of this kind are defined by extending the type processEntireEventType, as illustrated below.

```
enum MyEntrySignals : processEntireEventType {
      _PROCESS_ENTIRE_Signal_1,
      _PROCESS_ENTIRE_Signal_2
};
```

These signals must be derived from **processEntireEventType**. A process registers the acceptance of these signals via the **accepts**(…) specification at its **entry** points. The **entry** points of an application can have different **accepts**(…) lists, or none. However, depending on the **entry** point used to execute an application, its process will register the list for the start up **entry** point.

Below is an example of an entry point that registers two entire signals.

```
entry void main(void)
accepts(_PROCESS_ENTIRE_Signal_1, _PROCESS_ENTIRE_Signal_2)
      // Body of entry point
end;
```

Chapter VI.7. Extending Entire signals

Z++ applications can load other Z++ applications as their components. In this scenario, components become part of the process that loaded them. However, a Z++ application may also be designed as a set of communicating processes. In this section we illustrate the generalization of entire signals among the threads of a set of processes.

Observe that process-bounded entire signals do not cross the boundary of the process that generates them. That is, only the threads of the generating process can catch these signals. On the other hand, node-bounded signals only propagate among processes. Indeed, once a thread of a process catches these signals, no other thread of the same process will be able to catch it, anymore. Note that, multiple processes can register to catch node-bounded signals, though. What is missing is the ability for a process to have multiple of its threads catch a node-bounded entire signal. This is easily solved by, sequencing the two forms of entire signals.

In first step we define node-bounded entire signals for communication among processes. In second step, for each process we define its own process-bounded entire signals. When a thread of a process catches a node-bounded signal, it simply generates a process-bounded entire signal for use by its own threads.

The example of the next section illustrates this extension technique.

Remark. In general when a Z++ application is intended for use as a reusable component, its entry points should avoid registering signals. This is because the component may be loaded by applications that do not need to handle the signals. In other words, registration of entire signals by the entry points of a component limits its reusability by specializing it. When this the desired design for the use of a component, there are no issues or concerns. On the other hand, if signaling is the only means of communication among a set of cooperating Z++ components, it is better to load each component as a process. The writing of the programs for the components remains the same. However, instead of a top program loading the remaining programs as its components, each program is loaded as a separate process in its own right.

Section VI.7.A. Example for extending Entire Signals

The example of this section consists of three programs. These programs must run on Z47L (listening mode) as multiple processes under control of a single Z47 processor.

The **Z++ Internet Server** allows bringing up the Server for database, autonomous agent etc. **and** it also allows bringing up the Z47L Server in listening mode. Z47L is for loading multiple Z++ applications on a single Z47 processor, and for **delivering entire signals**, as the name listening mode indicates.

The executables of programs EntryOne.zpp and EntryTwo.zpp must be loaded before loading EntryCaller.zpp, so that the signal `_PROCESS_ENTIRE_Signal_1` gets registered before EntryCaller.zpp sends it out.

EntryCaller.zpp registers the signal `_PROCESS_ENTIRE_Signal_1` and then generates it. EntryOne.zpp and EntryTwo.zpp have also registered `_PROCESS_ENTIRE_Signal_1`.

So, all three programs will eventually receive (catch) `_PROCESS_ENTIRE_Signal_1.`

Each program, upon receiving `_PROCESS_ENTIRE_Signal_1` will generate the signal `_THREAD_ENTIRE_Signal_1`. This part of all three programs is identical for ease of understanding the technique. The signal `_THREAD_ENTIRE_Signal_1` is caught by, a **task** and a global **thread** in each of the programs. The simple signals are used for internal thread communication, in this case telling main() that the threads have completed their work, so main() can terminate.

```
// EntryOne.zpp

// ------------------------------------------------------------- //
// This program must be loaded on Z47L (listening mode)
// before EntryCaller.zpp.
// When loaded will wait for EntryCaller.zpp to load and send it
// the signal _PROCESS_ENTIRE_Signal_1.
// ------------------------------------------------------------- //
// After receiving _PROCESS_ENTIRE_Signal_1, the main() generates
// the entire signal _THREAD_ENTIRE_Signal_1 which is caught
// by a task object and a global thread.
// ------------------------------------------------------------- //

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////

enum MyOwnSignals : signalEventType {
      _SIGNAL_Thread_ended_1,
      _SIGNAL_Task_Handler_Ended_1
```

```
};

enum MyEntireSignals : threadEntireEventType {
      _THREAD_ENTIRE_Signal_1
};

enum MyEntrySignals : processEntireEventType {
      _PROCESS_ENTIRE_Signal_1
};

////////////////////////////////////////////////////////////////////
// The task registers to catch the entire
// signal _THREAD_ENTIRE_Signal_1.
// ---------------------------------------------------------------- //

task One

      accepts(_THREAD_ENTIRE_Signal_1);

      void HandlerOne(void)<_THREAD_ENTIRE_Signal_1>;

public:
end;

void One::HandlerOne(void)
      output << "EntryOne, Inside HandlerOne(void).";
      output << " Sending signal _SIGNAL_Task_Handler_Ended_1.\n";

      signal <- _SIGNAL_Task_Handler_Ended_1;
end;

////////////////////////////////////////////////////////////////////
// The global thread registers to catch the entire signal
// _THREAD_ENTIRE_Signal_1.
// ---------------------------------------------------------------- //

void GlobalThread(void)<thread>
      accepts(_THREAD_ENTIRE_Signal_1)

      output << "EntryOne, GlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_1.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

      output << "EntryOne, GlobalThread():";
      output << " sending _SIGNAL_Thread_ended_1.\n";

      signal <- _SIGNAL_Thread_ended_1;
end;

////////////////////////////////////////////////////////////////////
// main registers to catch _PROCESS_ENTIRE_Signal_1.
// ---------------------------------------------------------------- //
// When the signal _PROCESS_ENTIRE_Signal_1 arrives, main generates
// the entire signal _THREAD_ENTIRE_Signal_1 for its threads.
////////////////////////////////////////////////////////////////////
```

```
entry void main(void)
      accepts(_PROCESS_ENTIRE_Signal_1)

      output << "EntryOne: Hello World!\n";

      output << "EntryOne: Starting task and thread\n";

      One N;
      GlobalThread();

// Now wait until EntryCaller.zpp generates _PROCESS_ENTIRE_Signal_1.

      output << "EntryOne: Waiting for _PROCESS_ENTIRE_Signal_1.\n";
      do enddo(signal ? _PROCESS_ENTIRE_Signal_1);

// We have caught _PROCESS_ENTIRE_Signal_1. So now we generate
// _THREAD_ENTIRE_Signal_1 for task and global thread.

      output << "EntryOne: Sending _THREAD_ENTIRE_Signal_1.\n";

      signal <- _THREAD_ENTIRE_Signal_1;

// Wait until task handler completes its processing.

      output << "EntryOne:";
      output << " Waiting for _SIGNAL_Task_Handler_Ended_1.\n";

      do enddo(signal ? _SIGNAL_Task_Handler_Ended_1);

// Wait until global thread completes its processing.

      output << "EntryOne: Waiting for _SIGNAL_Thread_ended_1.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_1);

      output << "EntryOne: Good-bye World!\n"; // We are done.

end;
```

This is identical to EntryOne.zpp, except in its output it uses its own name. It must be loaded before EntryCaller.zpp.

```
// EntryTwo.zpp

// ------------------------------------------------------------------ //
// This program must be loaded on Z47L (listening mode)
// before EntryCaller.zpp.
// When loaded will wait for EntryCaller.zpp to load and send it
// the signal _PROCESS_ENTIRE_Signal_1.
// ------------------------------------------------------------------ //
// After receiving _PROCESS_ENTIRE_Signal_1, the main() generates
// the entire signal _THREAD_ENTIRE_Signal_1 which is caught by
// a task object and a global thread.
// ------------------------------------------------------------------ //
```

```
#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

//////////////////////////////////////////////////////////////////////

enum MyOwnSignals : signalEventType {
      _SIGNAL_Thread_ended_1,
      _SIGNAL_Task_Handler_Ended_1
};

enum MyEntireSignals : threadEntireEventType {
      _THREAD_ENTIRE_Signal_1
};

enum MyEntrySignals : processEntireEventType {
      _PROCESS_ENTIRE_Signal_1
};

//////////////////////////////////////////////////////////////////////
// The task registers to catch the entire
// signal _THREAD_ENTIRE_Signal_1.
// ------------------------------------------------------------------ //

task One

      accepts(_THREAD_ENTIRE_Signal_1);

      void HandlerOne(void)<_THREAD_ENTIRE_Signal_1>;

public:
end;

void One::HandlerOne(void)
      output << "EntryTwo, Inside HandlerOne(void).";
      output << " Sending signal _SIGNAL_Task_Handler_Ended_1.\n";

      signal <- _SIGNAL_Task_Handler_Ended_1;
end;

//////////////////////////////////////////////////////////////////////
// The global thread registers to catch the entire signal
// _THREAD_ENTIRE_Signal_1.
// ------------------------------------------------------------------ //

void GlobalThread(void)<thread>
      accepts(_THREAD_ENTIRE_Signal_1)

      output << "EntryTwo, GlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_1.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

      output << "EntryTwo, GlobalThread():";
```

```
      output << " sending _SIGNAL_Thread_ended_1.\n";

      signal <- _SIGNAL_Thread_ended_1;
end;

////////////////////////////////////////////////////////////////////
// main registers to catch _PROCESS_ENTIRE_Signal_1.
// ------------------------------------------------------------ //
// When the signal _PROCESS_ENTIRE_Signal_1 arrives, main generates
// the entire signal _THREAD_ENTIRE_Signal_1 for its threads.
////////////////////////////////////////////////////////////////////

entry void main(void)
      accepts(_PROCESS_ENTIRE_Signal_1)

      output << "EntryTwo: Hello World!\n";

      output << "EntryTwo: Starting task and thread\n";

      One N;
      GlobalThread();

// Now wait until EntryCaller.zpp generates _PROCESS_ENTIRE_Signal_1.

      output << "EntryTwo: Waiting for _PROCESS_ENTIRE_Signal_1.\n";
      do enddo(signal ? _PROCESS_ENTIRE_Signal_1);

// We have caught _PROCESS_ENTIRE_Signal_1. So now we generate
// _THREAD_ENTIRE_Signal_1 for task and global thread.

      output << "EntryTwo: Sending _THREAD_ENTIRE_Signal_1.\n";

      signal <- _THREAD_ENTIRE_Signal_1;

// Wait until task handler completes its processing.

      output << "EntryTwo:";
      output << " Waiting for _SIGNAL_Task_Handler_Ended_1.\n";

      do enddo(signal ? _SIGNAL_Task_Handler_Ended_1);

// Wait until global thread completes its processing.

      output << "EntryTwo: Waiting for _SIGNAL_Thread_ended_1.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_1);

      output << "EntryTwo: Good-bye World!\n"; // We are done.

end;
```

This program must be loaded last, after EntryOne.zpp and EntryTwo.zpp. It generates the signal _PROCESS_ENTIRE_Signal_1 that all three programs, including itself, will catch.

```
// EntryCaller.zpp

// ------------------------------------------------------------- //
// This program must be loaded on Z47L (listening mode).
// ------------------------------------------------------------- //
// This program must be loaded after EntryOne.zpp and EntryTwo.zpp.
// It is this program that sends the entire
// signal _PROCESS_ENTIRE_Signal_1.
// ------------------------------------------------------------- //
// This program itself registers to catch
// _PROCESS_ENTIRE_Signal_1, which it generates.
// ------------------------------------------------------------- //
// After receiving _PROCESS_ENTIRE_Signal_1, the main() generates
// the entire signal _THREAD_ENTIRE_Signal_1 which is caught by
// a task object and a global thread.
// ------------------------------------------------------------- //

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////

enum MyOwnSignals : signalEventType {
      _SIGNAL_Thread_ended_1,
      _SIGNAL_Task_Handler_Ended_1
};

enum MyEntireSignals : threadEntireEventType {
      _THREAD_ENTIRE_Signal_1
};

enum MyEntrySignals : processEntireEventType {
      _PROCESS_ENTIRE_Signal_1
};

////////////////////////////////////////////////////////////////////
// The task registers to catch the entire
// signal _THREAD_ENTIRE_Signal_1.
// ------------------------------------------------------------- //

task One

      accepts(_THREAD_ENTIRE_Signal_1);

      void HandlerOne(void)<_THREAD_ENTIRE_Signal_1>;

public:
end;
```

```
void One::HandlerOne(void)
      output << "EntryCaller, Inside HandlerOne(void).";
      output << " Sending signal _SIGNAL_Task_Handler_Ended_1.\n";

      signal <- _SIGNAL_Task_Handler_Ended_1;
end;

////////////////////////////////////////////////////////////////////
// The global thread registers to catch the entire signal
// _THREAD_ENTIRE_Signal_1.
// ---------------------------------------------------------------- //

void GlobalThread(void)<thread>
      accepts(_THREAD_ENTIRE_Signal_1)

      output << "EntryCaller, GlobalThread():";
      output << " waiting for _THREAD_ENTIRE_Signal_1.\n";

      do enddo(signal ? _THREAD_ENTIRE_Signal_1);

      output << "EntryCaller, GlobalThread():";
      output << " sending _SIGNAL_Thread_ended_1.\n";

      signal <- _SIGNAL_Thread_ended_1;
end;

////////////////////////////////////////////////////////////////////
// main registers to catch _PROCESS_ENTIRE_Signal_1. It then
// generates this signal which will be caught by
// EntryOne.zpp, EntryTwo.zpp and itself.
// ---------------------------------------------------------------- //
// When the signal _PROCESS_ENTIRE_Signal_1 arrives, main generates
// the entire signal _THREAD_ENTIRE_Signal_1 for its threads.
////////////////////////////////////////////////////////////////////

entry void main(void)
      accepts(_PROCESS_ENTIRE_Signal_1)

      output << "EntryCaller: Hello World!\n";

      output << "EntryCaller: Starting task and thread\n";

      One N;
      GlobalThread();

// ---------------------------------------------------------------- //
// Here, we generate _PROCESS_ENTIRE_Signal_1, which will be caught by
// all three processes: EntryOne.zxe, EntryTwo.zxe and EntryCaller.zxe.
// ---------------------------------------------------------------- //

      output << "EntryCaller: sending _PROCESS_ENTIRE_Signal_1.\n";
      signal <- _PROCESS_ENTIRE_Signal_1;

// ---------------------------------------------------------------- //
// Now we wait until _PROCESS_ENTIRE_Signal_1 is delivered.
// ---------------------------------------------------------------- //
```

```
      output << "EntryCaller: Waiting for _PROCESS_ENTIRE_Signal_1.\n";
      do enddo(signal ? _PROCESS_ENTIRE_Signal_1);

// We have caught _PROCESS_ENTIRE_Signal_1. So now we generate
// _THREAD_ENTIRE_Signal_1 for task and global thread.

      output << "EntryCaller: Sending _THREAD_ENTIRE_Signal_1.\n";

      signal <- _THREAD_ENTIRE_Signal_1;

// Wait until task handler completes its processing.

      output << "EntryCaller:";
      output << " Waiting for _SIGNAL_Task_Handler_Ended_1.\n";

      do enddo(signal ? _SIGNAL_Task_Handler_Ended_1);

// Wait until global thread completes its processing.

      output << "EntryCaller: Waiting for _SIGNAL_Thread_ended_1.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_1);

      output << "EntryCaller: Good-bye World!\n"; // We are done.

end;
```

The output is as follows.

Note. The blank lines are not part of the output. They have been added because the word processor has broken some lines, making the output difficult to understand.

```
EntryOne: Hello World!
EntryOne: Starting task and thread
EntryOne, GlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.
EntryOne: Waiting for _PROCESS_ENTIRE_Signal_1.
EntryTwo: Hello World!
EntryTwo: Starting task and thread
EntryTwo, GlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.
EntryTwo: Waiting for _PROCESS_ENTIRE_Signal_1.
EntryCaller: Hello World!
EntryCaller: Starting task and thread
EntryCaller, GlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.
EntryCaller: sending _PROCESS_ENTIRE_Signal_1.
EntryCaller: Waiting for _PROCESS_ENTIRE_Signal_1.
EntryTwo: Sending _THREAD_ENTIRE_Signal_1.
EntryTwo: Waiting for _SIGNAL_Task_Handler_Ended_1.
EntryOne: Sending _THREAD_ENTIRE_Signal_1.

EntryOne, Inside HandlerOne(void). Sending signal
_SIGNAL_Task_Handler_Ended_1.

EntryOne, GlobalThread(): sending _SIGNAL_Thread_ended_1.
```

```
EntryTwo, Inside HandlerOne(void). Sending signal
_SIGNAL_Task_Handler_Ended_1.

EntryTwo, GlobalThread(): sending _SIGNAL_Thread_ended_1.
EntryCaller: Sending _THREAD_ENTIRE_Signal_1.
EntryTwo: Waiting for _SIGNAL_Thread_ended_1.
EntryOne: Waiting for _SIGNAL_Task_Handler_Ended_1.
EntryOne: Waiting for _SIGNAL_Thread_ended_1.

EntryCaller, Inside HandlerOne(void). Sending signal
_SIGNAL_Task_Handler_Ended_1.

EntryCaller, GlobalThread(): sending _SIGNAL_Thread_ended_1.
EntryCaller: Waiting for _SIGNAL_Task_Handler_Ended_1.
EntryTwo: Good-bye World!
EntryOne: Good-bye World!
EntryCaller: Waiting for _SIGNAL_Thread_ended_1.
EntryCaller: Good-bye World!
```

Chapter VII. Distributed Signaling

In previous chapter we studied various forms of signaling within the scope of a single Z47 Processor, the local scope. However, Z47 is a distributed operating system. When a set of nodes cooperate in accomplishing a task, the distributed scope becomes the union of local scopes of the nodes. Nonetheless, Z++ language facilitates viewing the entire design of a distributed application as executing on a single node.

In this chapter we discuss Z++ model of distributed signaling. A distributed signal can be generated on one node and caught on another. Plain signaling does not convey sufficient information in a distributed scope. We shall illustrate how **Z++ distributed signals can carry along data to provide more information to the recipient**.

Chapter VII.1. Preliminaries

The notion of distributed signaling is identical to local signaling. Nonetheless, for clarity Z++ introduces new terms for distributed signaling. The sender of a distributed signal is said to **tell** the signal, rather than generate the signal. The recipient of a distributed signal is said to **hear** the signal, instead of catch the signal. Thus, we refer to distributed signals as tell-hear signals, or simply as either tell signals or hear signals. The operators remain the same as illustrate below.

As mentioned in previous chapter, all types of Z++ signals are defined in the header file exception.h and use the namespace exceptionSpace. The type of tell-hear signals is the enumeration type tellSignalType. For clarity, when extending these types it is recommended to prefix new enumeration literals with _TELL. Consider the following illustration, which extends tell-hear signals by adding three new signals.

```
#include <exception.h>
using namespace exceptionSpace;

enum MyTellSignals : tellSignalType {
      _TELL_telling_1,
      _TELL_telling_2,
      _TELL_telling_3
};
```

Local signals, discussed in previous chapter, used the following Z++ statements for generating and catching signals.

```
signal <- My_Signal; // generate a signal
signal ? My_Signal   // boolean expression for catching signal
```

The equivalent of above statements for tell-hear signals are as follows.

```
tell <- My_Signal; // generate a distributed signal
hear ? My_Signal   // boolean expression for catching signal
```

Tell-hear signals must be registered in the same manner as entire signals. However, instead of the statement **accepts**(…), distributed signals use **hears**(…). Note that the term for registering distributed signals is **hears**, while the primitive for catching them is **hear**.

Tell-hear signals are intended for **distributed** signaling. A process on one node tells a signal, another process on another node hears it. Nonetheless, since these signals can also carry data, Z++ allows their use among processes on a single node, and even among threads of the same process.

Chapter VII.2. Acceptance and Rejection of Tell signals

Consider a Z47 **source** node (the node on which a process tells a signal), and a Z47 **destination** node (the node on which a process will hear the signal). When the source node tells a signal, the destination node checks to see if any process within its local scope has registered to hear that signal. If the signal has been registered, the destination node will queue the signal for later delivery. In this case, the destination node has **accepted** the signal and sends the acknowledgment of its acceptance to the source node. On the other hand, if no process on the destination node has yet registered to hear the signal, the destination node **rejects** the signal and informs the source node of its rejection. When the source node receives rejection from destination node, it raises the following exception.

```
_EXCEPTION_SIGNAL_SignalNotRegistered
```

It is up to the process to try another destination, or to take another suitable action. Note that the statement that might raise the exception is the following.

```
tell <- My_Signal;
```

In contrast, the following equivalent of the above statement for signals within the local scope of Z47 does not raise exceptions.

```
signal <- My_Signal;
```

Chapter VII.3. Data transmitted with Tell signals

Distributed tell-hear signals are capable of carrying data along with signals. A hear signal is identified with the signal itself and its **signature**. The signature of a signal is the list of types of data associated with it in a specific order, just as a function signature. The signature of a signal without data is implicitly understood as **void**.

A hear signal prototype is similar to a function prototype, as illustrated below.

```
My_Signal< type_1, type_2, ...>
```

There are no formal parameters. The ellipses mean that a prototype can have more types specified, and are **not** part of the syntax.

Chapter VII.4. Registering Hear signals

A signal prototype is only used during registration. A registration statement looks like follows.

```
hears(_sig < type_1, type_2, ... > , ... )
```

The second ellipses mean more **hear** signal prototypes.

We will illustrate the telling (sending) a signal later. The syntax for hearing (catching) a signal with **void** signature was illustrated earlier as follows.

```
hear ? My_Signal
```

When the signature is not **void**, the catching of the signal completes when the data associated with the signal have been copied. We will illustrate this shortly. The syntax is as follows.

```
hear ? My_Signal : object_1, Object_2, ...
```

That is, a colon is used as a separator, followed by objects to be copied over.

Chapter VII.5. Registration for Global Threads

We will use simple data types to illustrate the registration of **hear** signals. However, **tell** signals can have complex signature, including user-defined **class** types. For illustrations, we shall use the following signals, defined earlier in this chapter.

```
enum MyTellSignals : tellSignalType {
      _TELL_telling_1,
      _TELL_telling_2,
      _TELL_telling_3
};
```

Below is a global thread registering two **hear** signal prototypes.

```
void hearFun(void)<thread>
hears(_TELL_telling_1<int, double>, _TELL_telling_2)

// Define local objects

      int i;
      double d;

// Wait until the signal is caught

      do enddo(hear ? _TELL_telling_1 : i, d);

// Handle the signal
// Now wait until the next signal is caught

      do enddo(hear ? _TELL_telling_2);

// Handle the signal

end;
```

The signature of `_TELL_telling_1` is **int** and **double** in that order. The signature of the second signal `_TELL_telling_2` is **void**.

Notice that when the signal `_TELL_telling_1` is caught the local objects i and d are also copied over. The values came along from the **teller** of the signal.

Chapter VII.6. Registration for Task Threads

As with **accepts**(…) specification for local signals, a **task** type specifies **hears**(…) statements in its definition, preferably in its **private** section for readability.

```
task MyTask

      int ti;
      double di;

      hears(_TELL_telling_1<int, double>, _TELL_telling_2);

public:
end;
```

The signal prototypes are the same as in the example for the global thread in previous section.

Chapter VII.7. Task Handlers

The Z++ compiler will match each registered signal with a handler. That is, you will need to provide a handler for each registered signal. Below, we repeat the above task definition, adding handlers to it.

```
task MyTask

      int ti;
      double di;

      hears(_TELL_telling_1<int, double>, _TELL_telling_2);

      void FirstHearHandler(void)<_TELL_telling_1 : ti, di>;
      void SecondHearHandler(void)<_TELL_telling_2>;

public:
end;

// Definition of handlers
void MyTask::FirstHearHandler(void)
      // Body of handler
end;

void MyTask::SecondHearHandler(void)
      // Body of handler
end;
```

Task handlers cannot be invoked directly. Instead, **task** object invokes the attached handler when Z47 informs it of the arrival of a signal. Furthermore, the body of handler **does not include a catch statement for its signal**. The catching of the signal is done by, the **task** object prior to invoking its attached handler.

You will notice one difference in the specification of **task** handlers for **hear** signals, compared to **entire** signals. The handler `FirstHearHandler` lists members ti and di of the **task** type after the separator colon. When the **task** object catches the signal `_TELL_telling_1` it also receives the **data** associated with the signal from the teller of the signal. **It is important to remember that task object copies the received data to the members listed in the specification of a handler before invoking the handler**.

Chapter VII.8. Telling Distributed signals

The teller (sender, generator) of a signal must include objects of appropriate types along with the signal. For instance, for the signal `_TELL_telling_1` mentioned in previous section (signature **int**, **double**) the teller must include two objects as follows.

```
int ii;
double dd;

tell <- _TELL_telling_1 : ii, dd;
```

When the above statement is executed, Z47 will send the signal and the values of objects ii and dd together.

Tell-hear signals are intended for distributed signaling. That means, the syntax of tell statement must allow for specifying other nodes, and possibly specific processes on those nodes. The above tell statement is for use within the local scope of Z47 Processor. That is, the sender and the receiver of the signal are processes on the same node. In fact, the sender and the receiver could be the same process. Remember that in order to hear (catch) a tell-hear signal, a thread of the process must register the signal.

The full syntax of a tell statement is as follows.

```
tell <- sig $ URL , process : object_1, object_2, ... ;
```

The $ sign is a separator indicating that what follows is a string for the IP-Address of the node to receive the signal. This is followed with the comma separator and another string for the name of the process to be notified of the arrival of the signal. Finally, the separator colon is followed by the list of objects to be sent along with the signal.

Since everything starting with the $ is optional, the syntax may take one of the following forms.

When the URL and process name are not present, but the signal is intended to deliver data on the local node, the syntax takes the following form.

```
tell <- sig : object_1, object_2, ... ;
```

When there is a URL, with and without data, it takes the following forms.

```
tell <- sig $ URL : object_1, object_2, ... ;
tell <- sig $ URL;
```

Without a URL, but with a process name (local scope), it takes the following forms.

```
tell <- sig , process : object_1, object_2, ... ;
tell <- sig , process;
```

From these it should be clear why the separator $ was chosen instead of a second comma.

The tell statement could **raise** the following exceptions, in which case the receiving end (destination) has **rejected** the signal. Note that when the destination node rejects a signal nothing happens on the destination node. The exceptions are raised on the source node, the node on which the **tell** statement was executed.

1. `_EXCEPTION_SIGNAL_SignalNotRegistered`. This exception is raised when the destination node informs the sender that no process has yet registered to **hear** the signal.
2. `_EXCEPTION_SIGNAL_TellOperandTypeMismatch`. In this case, the destination node has responded that, though the signal has been registered, the types of data received along with the signal do not match the types specified in **hears**(…) statement that has registered this signal.

From the second exception above it can be seen that some negotiation takes place prior to sending the data associated with a distributed signal. The teller first sends the signature of the signal to the destination node for verification. It packages and transmits the data **only if the destination node accepts the signal**.

Chapter VII.9. Further support for Tell-Hear signals

In an entire application (running as a process) hear signals are unique. Compiler will not allow the same **hear** signal to be registered more than once. Thus at runtime, exactly one thread in a process will register each **hear** signal.

Hear expressions for a signal can only appear in the same thread that has registered the signal. This is enforced by, the compiler. Therefore, if at runtime several **tell** data arrive for a signal before the matching **hear** expressions are executed, they will be delivered in the order in which the **tell** statements were executed. No signals are lost.

When the process name is not specified in a **tell** statement, Z47 Processor will deliver the signal to the first process that it locates as the one that has registered for the signal. Hear signals are **not entire**. Once caught, they are removed from the queue.

Tell/hear distributed signaling is for inter-process communication, particularly between processes on different nodes. However, they can also be used within a single process. In the latter case care is needed with regard to objects passed in a **tell** statement. If the **tell** statement is in a function, and uses local objects as operands to **tell**, those objects will be destroyed as soon as the function ends. However, the signal may not have been delivered yet. It is better to use **global** objects in this scenario, or make sure the function executing the **tell** statement does not end too soon. This is so because in case of signaling within a **single** process, Z++ compiler uses references to the data sent along with a **tell** statement. In all other cases **objects are actually copied**, and there is no concern.

Let us explain this with an example. Consider two pilots engaged in accomplishing a certain task. When these pilots reach their target, they will be communicating back and forth in order to synchronize their operations. The pilots will benefit from data arriving from a central command. However, it is their ability to communicate with one another that helps them accomplish the task, successfully.

The points of interest to observe from the above example are as follows. First, the pilot that begins a communication cannot freeze in the sky until he hears back. Second, the pilot that is addressed must make adjustments based on his current state because he too is not frozen in the sky as he hears from his friend. It may now be clear while the model is called Communicating Concurrent Processes (CCP).

As another example, robots in a car factory can synchronize their local operations more smoothly by communicating with one another rather than a central computer. This model can be applied to any large system that comprises of several subsystems each of which is running independently under the control of its own program. The subsystems may need to synchronize their operations in order for the entire system to operate smoothly. The central command is needed for major overall decisions. Local operations require synchronization among the entities attempting to accomplish an assigned task.

Tell-Hear signaling provides a model for CCP.

Chapter VII.11. Web Services

The tell-hear distributed signaling can also be viewed as another abstraction, the Asynchronous Function Call (AFC). A Remote Procedure Call (RPC) is a function call, which blocks the caller until the body of the function is executed. In contrast, a **tell** signal acts just like a function call without blocking. Now, since the response of the call will also arrive via the **tell** mechanism, the entire call therefore was made more like **sending an email and at a later time receiving an email back**.

When viewed as AFC, a **tell signal** is the name of the function, and the arguments sent along the signal are the arguments to the call. The sender of response uses the same mechanism. However, from the perspective of the first caller, it is simply receiving the reply. This is no different than sending and receiving emails.

The data sent along with a **tell** signal must match the same pattern as in a function call. That is, the types and the number of arguments are checked just like a function call, thus the reason for nomenclature.

The AFC in the hands of those interested in solving problems will find many uses. Here we describe a solution for a contemporary problem, the **Web Services**.

In an RPC, it is expected for the call to go through. For that reason, the actual data is sent along with the call. Z47 Processor follows a different protocol when dealing with **tell** signal. Initially, it sends the signal and the types of the data, but not the actual data. It may receive an acceptance or a rejection from the destination Z47. **It only sends the data in binary format if it receives acceptance for the service.** The destination Z47 sends an acceptance only if a process has registered for receiving the signal with exact types of data associated with that signal.

It should be clear that the destination server could itself seek for the service by contacting other servers before rejecting the requested service from a client. One of the obstacles of setting up servers for Web Services has been the negotiation for the type, number and order of arguments. The Z++ **tell** signaling makes this issue entirely transparent to the engineer writing a program that needs to look for certain services on the web.

In order for tell signaling to work smoothly for purposes of web services a pattern similar to ports is to be followed. For instance, suppose vender XYZ has offered a few services. XYZ will make an include file as follows.

```
// XYZ_Services.h

#include<exception.h>
using namespace exceptionSpace;

enum XYZ_Service_Signals : tellSignalType {
      _SIGNAL_telling_XYZ_ServiceOneName,
      _SIGNAL_telling_XYZ_ServiceTwoName,
```

```
      _SIGNAL_telling_XYZ_ServiceThreeName,
      _SIGNAL_telling_XYZ_ServiceFourName
};
```

Now, suppose another vender UVW has new services to publish. UVW will include the file XYZ_Services.h, and create the following include file.

```
// UVW_Services.h

#include< XYZ_Services.h>

enum UVW_Service_Signals : XYZ_Service_Signals {
      _SIGNAL_telling_UVW_ServiceOneName,
      _SIGNAL_telling_UVW_ServiceTwoName,
      _SIGNAL_telling_UVW_ServiceThreeName,
      _SIGNAL_telling_UVW_ServiceFourName
};
```

The next vender will follow the same pattern as UVW. This will ensure that all web services are **unique throughout the Web**. Now, we only need a specific site to place all header files, perhaps with description of services. This will allow users to locate the service they are looking for, and providers to know which is the last header file that they need to include for their services before publishing them.

There is no point is assigning numeric values to enumeration literals. In fact, if two **tell** signals provided as services by two different venders branch off of system **tell** signals `tellSignalType` the numeric integer values will have no effect. The services will become indistinguishable. As an example consider the following.

```
enum XYZ_Service_Signals : tellSignalType {
      _SIGNAL_telling_XYZ_ServiceOneName,
      _SIGNAL_telling_XYZ_ServiceTwoName,
      _SIGNAL_telling_XYZ_ServiceThreeName,
      _SIGNAL_telling_XYZ_ServiceFourName
};

enum UVW_Service_Signals : tellSignalType {
      _SIGNAL_telling_UVW_ServiceOneName,
      _SIGNAL_telling_UVW_ServiceTwoName,
      _SIGNAL_telling_UVW_ServiceThreeName,
      _SIGNAL_telling_UVW_ServiceFourName
};
```

If these are included in one project, there is no problem. However, when two different venders use the above pattern, Z47 will see their services as identical. For instance, Z47 will see the **tell** signal `_SIGNAL_telling_XYZ_ServiceOneName` as identical to the **tell** signal `_SIGNAL_telling_UVW_ServiceOneName`.

Chapter VII.12. Tell-Hear Example

Z++ Internet Server has multiple functions. For tell/hear distributed signaling the program that registers **hear** signals must run on Z47L, the Z47 in listening mode. The Z++ Internet Server allows bringing up the Internet Server for Database, Autonomous Agent and Remote Components etc. The Internet Server also allows bringing up the Z47 in listening mode. This allows loading and executing multiple Z++ programs under a single Z47. A Z++ program that registers hear signals must be loaded for execution in Z47 listening mode.

Remark. Z++ Internet Server also allows running a Z++ program as standalone, under its own Z47 processor. The program that registers hear signals should not be executed as standalone. Instead, it must be executed in multi-process mode (Z47L). However, any other program, including the one that sends out tell signals, can be executed as standalone or in multi-process mode.

The types of data allowed for tell-hear distributed signaling include: all numeric types from **char** to **double**, **string** and **enumeration**, as well as **class**es of these types. Classes can have **class members** and could be **derived** from other classes.

In this chapter we provide a complete tell-hear example. There are three files for this example. The first file, TellHear.h is a file included in the next two files.

The file Hear.zpp is the program that registers to **hear** signals, and must be loaded in Z47 listening mode (multi-process mode).

The file Tell.zpp is the program that sends **tell** signals to the executable of previous program Hear.zxe.

Below is the include file that defines data types used in tell/hear communication.

```
// TellHear.h

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

//-------------------------------------------------------------------//

enum MyOwnSignals : signalEventType {
      _SIGNAL_Thread_started,
      _SIGNAL_Thread_ended
};

enum MyTellSignals : tellSignalType {
      _SIGNAL_telling_1,
      _SIGNAL_telling_2,
```

```
	_SIGNAL_telling_3,
	_SIGNAL_telling_4,
	_SIGNAL_telling_5,
	_SIGNAL_telling_6,
	_SIGNAL_telling_7
};

// ------------------------------------------------------------------ //

enum simpleEnum {
	_SIMPLE_enum_1,
	_SIMPLE_enum_2,
	_SIMPLE_enum_3
};

// ------------------------------------------------------------------ //

struct argumentType
	int n;
	double d;

	argumentType(void);
	argumentType(int, double);
end;

argumentType::argumentType(void)
	n = 2;
	d = 8.9;
end;

argumentType::argumentType(int nn, double dd)
	n = nn;
	d = dd;
end;

// ------------------------------------------------------------------ //

struct imbeddedType
	simpleEnum e;
	string s;

	imbeddedType(void);
	imbeddedType(simpleEnum, string);
end;

imbeddedType::imbeddedType(void)
	e = _SIMPLE_enum_1;
	s = "I am member of imbeddedType";
end;

imbeddedType::imbeddedType(simpleEnum ee, string ss)
	e = ee;
	s = ss;
end;

// ------------------------------------------------------------------ //
```

The following program registers hear signals. Its executable, which we call Hear.zxe must run on a **Z47 in listening mode**.

```
// Hear.zpp

#include "~/TellHear.h" // use correct path to reach the file

//-------------------------------------------------------------------//
// The thread that registers to accept hear signals

void hearFun(void)<thread>
      hears(
      _SIGNAL_telling_3<argumentType>,
      _SIGNAL_telling_4<simpleEnum, string>,
      _SIGNAL_telling_5<imbeddedType>)

// ********************************************************** //
// Hearing _SIGNAL_telling_3.

      argumentType at; // Object for receiving data

// Wait until the signal and its data arrive.

      do enddo(hear ? _SIGNAL_telling_3 : at);

  output << "(hearFun) Values received for _SIGNAL_telling_3 ...\n";

      output << "argumentType.n is: " << at.n << '\n';
      output << "argumentType.d is: " << at.d << '\n';

      output.flush();

// ********************************************************** //
// Hearing _SIGNAL_telling_4.

// Declare object for receiving data

      simpleEnum se;
      string s;

// Wait until the signal and its data arrive.

      do enddo(hear ? _SIGNAL_telling_4 : se, s);

  output << "(hearFun) Values received for _SIGNAL_telling_4 ...\n";

      output << "simpleEnum is: " << [se] << '\n';
      output << "string is: " << s << '\n';

      output.flush();

// ********************************************************** //
// Hearing _SIGNAL_telling_5.

      imbeddedType it; // Object for receiving data
```

```
// Wait until the signal and its data arrive.

      do enddo(hear ? _SIGNAL_telling_5 : it);

  output << "(hearFun) Values received for _SIGNAL_telling_5 ...\n";

      output << "imbeddedType.e is: " << [it.e] << '\n';
      output << "imbeddedType.s is: " << it.s << '\n';

      output.flush();

// ************************************************************** //
// Inform main we are done ====

      signal <- _SIGNAL_Thread_ended;

end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hearer: Hello World!\n";

// Begin the thread that registers to hear signals

      hearFun();

// Wait until the above thread tells us it has received all
// tell-hear signals it registered.

      do enddo(signal ? _SIGNAL_Thread_ended);

      output << "Hearer: Good-bye World!\n";

end;
```

The following program is the one that sends **tell** signals to Hear.zxe. It can execute in any mode, from anywhere. That is, it can be loaded on same node as Hear.zxe in multi-process mode, as a standalone program from a remote node, etc.

```
// Tell.zpp

#include "~/TellHear.h" // use correct path to reach the file

////////////////////////////////////////////////////////////////////
// The Z++ Internet Server IP is the address of the node on which
// Hear.zxe is running in multi-process mode (Z47L). Put correct
// IP address below before compiling this program.
//----------------------------------------------------------------//

string url = "192.168.1.64"; // Z++ Internet Server IP
string process = "Hear.zxe"; // Process name to receive signals

////////////////////////////////////////////////////////////////////
//----------------------------------------------------------------//
////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Teller: Hello World!\n";

// Define data to send along with tell signal.

      argumentType at(777, 22.33);

// Send the signal and data to Hear.zxe, at node URL

      tell <- _SIGNAL_telling_3 $ url, process: at;

// Define data to send along with tell signal.

      string s = "I am sent by teller.";
      simpleEnum se = _SIMPLE_enum_2;

// Send the signal and data to Hear.zxe, at node URL

      tell <- _SIGNAL_telling_4 $ url, process: se, s;

// Define data to send along with tell signal.

      se = _SIMPLE_enum_3;
      imbeddedType it(se, "Teller string.");

// Send the signal and data to Hear.zxe, at node URL

      tell <- _SIGNAL_telling_5 $ url, process: it;

      output << "Teller: Good-bye World!\n";
end;
```

The output of Hear.zxe is as follows. This will show on Z++ Internet Server window.

```
Hearer: Hello World!
(hearFun) Values received for _SIGNAL_telling_3 ...
argumentType.n is: 777
argumentType.d is: 22.33
(hearFun) Values received for _SIGNAL_telling_4 ...
simpleEnum is: 1
string is: I am sent by teller.
(hearFun) Values received for _SIGNAL_telling_5 ...
imbeddedType.e is: 2
imbeddedType.s is: Teller string.
Hearer: Good-bye World!
```

Chapter VII.13. Task Tell-Hear Example

The example of this chapter consists of two programs communicating via tell-hear distributed signaling. The first program, TaskHear.zpp uses **task** handlers for responding to **hear** signals, and the second program, ThreadHear.zxe uses a global **thread**.

Both programs must be loaded via Z47L (multi-process listening mode). They can be loaded on same node, or two physically distinct nodes. However, the executable of TaskHear.zpp must be loaded **first** in order to avoid loss of signal and the raising of exception. That is because TaskHear.zpp, after registering its **hear** signals, begins with listening. On the other hand, ThreadHear.zpp first sends its **tell** signals before listening for arriving **hear** signals. That is a natural scenario in a conversation. One speaks and the other responds.

```
// TaskHear.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

// This program must start first on a Z47L. It registers hear signals
// and immediately begins listening.

////////////////////////////////////////////////////////////////////
// Make new signals by extending corresponding system signals.
// ---------------------------------------------------------------- //

enum MyFirstSignals : signalEventType {
      _SIGNAL_TaskAllDone,
      _SIGNAL_Terminate
};

enum MyTellSignals : tellSignalType {
      _SIGNAL_telling_1,
      _SIGNAL_telling_2,
      _SIGNAL_telling_3,
      _SIGNAL_telling_4,
      _SIGNAL_telling_5
};

// Use correct ip address to reach the desired node.

string url = "xxx.xxx.xxx.xxx";

////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //

task HearingTask
```

```
// Objects for receiving hear signals

      int intHear;
      double dblHear;
      string stgHear;

// Register tell/hear signals.

      hears(_SIGNAL_telling_1<int, double>, _SIGNAL_telling_2<string>);

// Handler for tell/hear signals.

  void FirstHearHandler(void)<_SIGNAL_telling_1 : intHear, dblHear>;
  void SecondHearHandler(void)<_SIGNAL_telling_2 : stgHear>;

public:

      HearingTask(void);

end;

// ------------------------------------------------------------------ //

HearingTask::HearingTask(void)
      intHear = 3;
      dblHear = 5.2;
      stgHear = "Initial Value";
end;

void HearingTask::FirstHearHandler(void)
      output << "Inside FirstHearHandler(void).\n";

      output << "FirstHearHandler got: " << intHear << ' ' <<
            dblHear << '\n';

      tell <- _SIGNAL_telling_3 $ url : "FirstHearHandler";
end;

void HearingTask::SecondHearHandler(void)
      output << "Inside SecondHearHandler(void).\n";
      output << "SecondHearHandler got: " << stgHear << '\n';

      tell <- _SIGNAL_telling_4 $ url : "SecondHearHandler";

// Inform main we are all done.

      signal <- _SIGNAL_TaskAllDone;
end;

//////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
//////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Task: Hello World!\n";

// Create a task object
```

```
      HearingTask HT;

// Wait for task object to complete its communication

      output << "Main: Waiting for _SIGNAL_TaskAllDone....\n";
      do enddo(signal ? _SIGNAL_TaskAllDone);

      output << "Task: Good-bye World!\n";
end;
```

This program uses a global thread in its tell-hear communication with TaskHear.zpp. It must begin execution after TaskHear.zpp has already started.

```
// ThreadHear.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

// ------------------------------------------------------------------ //
// The program must run in Z47L.
// ------------------------------------------------------------------ //
// If this program begins execution before TaskHear.zpp,
// we will get exception of signal not registered. That is
// because the main immediately sends signal _SIGNAL_telling_1.
// ------------------------------------------------------------------ //

//////////////////////////////////////////////////////////////////////
// Make new signals by extending corresponding system signals.
// ------------------------------------------------------------------ //

enum MyFirstSignals : signalEventType {
      _SIGNAL_TaskAllDone,
      _SIGNAL_Terminate
};

enum MyTellSignals : tellSignalType {
      _SIGNAL_telling_1,
      _SIGNAL_telling_2,
      _SIGNAL_telling_3,
      _SIGNAL_telling_4,
      _SIGNAL_telling_5
};

// Use correct ip address to reach the desired node.

string url = "xxx.xxx.xxx.xxx";

// ------------------------------------------------------------------ //
// The global thread that registers to hear signals.
```

```
void hearFun(void)<thread>
      hears(
      _SIGNAL_telling_3<string>,
      _SIGNAL_telling_4<string>)

      string HearObject; // Object for receiving hear data

// ************************************************************** //
// Hearing _SIGNAL_telling_3.

      do enddo(hear ? _SIGNAL_telling_3 : HearObject);

      output << "(hearFun) Heard _SIGNAL_telling_3 ...\n";
      output << "String received is: " << HearObject << '\n';

// ************************************************************** //
// Hearing _SIGNAL_telling_4.

      do enddo(hear ? _SIGNAL_telling_4 : HearObject);

      output << "(hearFun) Heard _SIGNAL_telling_4 ...\n";
      output << "String received is: " << HearObject << '\n';

// ************************************************************** //
// Inform main we are all done.

      signal <- _SIGNAL_Terminate;

end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      hearFun(); // Begin global thread for hearing

// Objects to send along tell signal

      int intObject = 97;
      double dblObject = 47.77;

// Send tell signals to TaskHear.zpp

      tell <- _SIGNAL_telling_1 $ url : intObject, dblObject;

      tell <- _SIGNAL_telling_2 $ url : "I am waiting.";

      output << "Waiting for hearFun() thread to end.\n";

// Wait until global thread completes.

      do enddo(signal ? _SIGNAL_Terminate);
```

```
        output << "Good-bye World!\n";
end;
```

Output for TaskHear.zpp and ThreadHear.zpp is as follows.

Note. The output is for running both programs on a single Z47 as communicating processes. The two programs were load in Z47L multi-process listening mode. So, the output is intermixed depending on the time-slice of executing threads.

```
Task: Hello World!
Main: Waiting for _SIGNAL_TaskAllDone....
Hello World!
Waiting for hearFun() thread to end.
Inside FirstHearHandler(void).
FirstHearHandler got: 97 47.77
(hearFun) Heard _SIGNAL_telling_3 ...
String received is: Inside SecondHearHandler(void).
SecondHearHandler got: FirstHearHandler
I am waiting.
(hearFun) Heard _SIGNAL_telling_4 ...
String received is: Task: Good-bye World!
SecondHearHandler
Good-bye World!
```

Chapter VIII. Exception Mechanism

An exceptional situation occurs when the execution of a statement cannot proceed due to circumstances that invalidate the conditions necessary for normal execution of the statement. In response to such an event Z47 **implicitly raises** an exception. At this point, the execution of the statement is interrupted and control is transferred to the **handler** specified for the **raised** exception. If a process has not specified a handler for a raised exception, Z47 will **terminate** the execution of the process.

Processes can also **explicitly** request Z47 to raise exceptions on their behalf. Whether an exception is raised, implicitly or explicitly, Z47 will always look for a handler **before** terminating a process. However, certain exceptions in a multi-user system terminate the process without any form of signaling to the process, such as illegal memory access in UNIX. This is necessary for security matters. In such cases, Z47 cannot inform its processes of the occurrence of an exception.

Z++ linguistic mechanism for the notion of exception uses **enumeration**. Exceptions implicitly raised by Z47 are the values of the enumeration type exceptionEventType. The type exceptionEventType is defined in the header file exception.h. We refer to these predefined exceptions as the **system exceptions**.

Since enumeration type constructor is extensible, so are system exceptions. Engineers define new **user exceptions** by extending system exceptions. Some user exceptions may also be raised implicitly, like when **class** invariants are violated. However, in this case the Z++ **compiler** arranges for raising those exceptions on behalf of a process. **Z47 only raises system exceptions**.

The Z++ linguistic mechanism for establishing a layer for exception handling is the **layer** statement. Furthermore, Z++ provides two forms of resumption from exception, **resume** and **repeat**.

Chapter VIII.1 Extending System Exceptions

New user-defined exceptions are introduced by extending the system exceptions. The system exceptions use the namespace `exceptionSpace`. In the example below, we are extending the system exceptions `exceptionEventType` to `userExceptions`, which introduces two new exceptions. It is recommended to use the prefix _EXCEPTION for user-defined exception literals, as is done for system exceptions.

```
#include<exception.h>
using namespace exceptionSpace;

enum userExceptions : exceptionEventType
{
      _EXCEPTION_FirstUserException,
      _EXCEPTION_SecondUserException
};
```

Chapter VIII.2 Raising Exceptions

Any exception can be raised explicitly, including system exceptions. However, one should define new exceptions with names that indicate the kind of problem that might have occurred, and may need attention. It is better to leave system exceptions for the system in order to avoid the confusion as to why an exception was raised.

The statement for explicitly raising an exception is **raise**. For instance, the following statement raises a user exception defined in previous chapter.

```
raise(_EXCEPTION_FirstUserException);
```

Chapter VIII.3 The structure of Layer Statement

The **layer** statement consists of two sections: guarded and handling. The guarded section is also called the **body** of the **layer** statement. When an exception occurs in the body of a **layer** statement, control is sent to its handling section. A **layer** statement requires the type of exceptions that can be handled within its scope.

The figure below shows the sections of a **layer** statement, using the exception type `userExceptions` defined above, in Chapter VIII.1.

```
layer<userExceptions>

// body of layer

handler

// handling section of layer

endlayer;
```

This **layer** statement will only allow defining handlers, in its handling section, for the exception type `userExceptions`. The type of an exception **layer** is significant with regard to the use of **else** and **elsall**, as we shall illustrate in later chapters.

If execution reaches end of body of a **layer** statement, without any exception being raised, control is transferred to the statement following **endlayer**.

When an exception occurs in the body of a **layer** statement for which a handler is found in the handling section, control is transferred to the **handler** for that exception. After the completion of the execution of the handler, and in the absence of resumption, control is transferred to the statement following **endlayer** for normal execution.

All objects created in the body of a **layer** statement are **destroyed** at **endlayer**. Thus, all these objects are available in the handling section of a **layer** statement.

If no handler is found for an exception raised (implicitly or explicitly) in the body of a **layer** statement, control goes to **endlayer**. After the **destruction** of objects created in the body of the **layer** statement, control will be sent to the handling section of the next outer **layer** statement. If after exhausting all **layer** statements no handler is found, Z47 will **terminate** the execution of the process.

If an exception occurs during the execution of a function called in the body of a **layer** statement, all objects created in that function until the point at which the exception occurred are **destroyed**, and the **call-stack** information for that function is cleaned up. This is done recursively for a nested sequence of function calls.

Chapter VIII.4. Handling Exceptions

Handlers are defined in the handling section of a layer statement. Using the exception type `userExceptions` defined earlier in Chapter VIII.1, the following figure illustrates the pattern.

```
layer<userExceptions>

// body of layer

handler

      case _EXCEPTION_FirstUserException:
            // body of handler

      case _EXCEPTION_SecondUserException:
            // body of handler

endlayer;
```

A **handler** starts with a **case** as in **switch** statement. Leaving the body of the handler blank is equivalent to **ignoring** the exception without any handling. This is only useful in initial step of development until one figures out how to handle an exception.

As seen in the figure, the handling section is essentially a **switch** on the type of the exception of the **layer** statement. Therefore, you can provide handlers for base types of the type of the **layer** statement. However, you cannot specify handlers for other kinds of exceptions in the handling section of the **layer** statement.

Chapter VIII.5. Else and Elsall

There are times that we want to handle several exceptions with a single handler. Since handlers are **case** legs, we can do so using range and sequence of cases as in **switch** statement. However, even **switch** statement allows an **else** leg.

The meaning of **else** leg in a **switch** statement is that, when none of the **case** legs is selected for execution, control is transferred to the **else** leg. However, in a **layer** statement the meaning of **else** is relative to the type of the **layer** statement. Consider the following.

```
layer<userExceptions>

// body of layer

handler

      case _EXCEPTION_FirstUserException:
            // body of handler

      else
            // body of else handler

endlayer;
```

The above **layer** statement is identical to the one in previous chapter. The **else** handler will only catch the exception `_EXCEPTION_SecondUserException`. In other words, the **else** leg will catch all the exceptions of the type of **layer** statement for which a handler has not been provided.

Now, `userExceptions` is derived from the system exceptions `exceptionEventType`, and yet the **else** leg will not catch system exceptions like `_EXCEPTION_DivisionByZero`. To catch all exceptions in all bases of the **layer** type and not, use **elsall**. In the figure below, the **elsall** leg will also catch `_EXCEPTION_DivisionByZero`. Note that even **elsall** cannot catch native operating system exceptions, such as segmentation fault in Linux. Such exceptions are not reported to the process that caused them. The native operating system simply terminates the process.

```
layer<userExceptions>
// body of layer
handler

      case _EXCEPTION_FirstUserException:
            // body of handler

      elsall
            // body of else handler
endlayer;
```

Chapter VIII.6. Resumption

It should be clear that **exception is not equivalent to termination**. A process could simply be terminated without raising any exceptions. The utility of exception mechanism is precisely in its ability to allow **resumption**.

Z++ provides two forms of resumption: **repeat** and **resume**. The example illustrates **repeat** and **resume** statements. The meaning of **repeat** is that, **after handling** the exception we wish to go back to the **statement that caused the exception**, and **repeat** its execution. However, **resume** goes back to the statement following the one that caused the exception, and **resumes** execution.

In the example there are two **layer** statements. The second **layer** statement is nested inside the outer **layer** statement, in fact in one of its handlers. The inner **layer** statement illustrates the use of **resume**, while the outer **layer** statements illustrates the use of **repeat**.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
// A function for raising exception _EXCEPTION_DivisionByZero.
//--------------------------------------------------------------------//

int globalFun(int f, int s)
      f /= s; // raises exception when s == 0
      return 47;
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      int i = 10;
      int j = 0;

      layer<exceptionEventType>

            output << "Starting layer.\n";

            int ret = 777;

// Next statement will cause _EXCEPTION_DivisionByZero to happen.
// Then, repeat will come back here and make the function call again.
```

```
            ret = globalFun(i, j);

 // Will not execute first round

            output << "ret is: " << ret << '\n';

      handler

            case _EXCEPTION_DivisionByZero:

// This is a nested layer inside of a case leg.
// resume and repeat within each handler go to the body
// of the layer containing them.

                  int k;

                  layer<exceptionEventType>

                        k = 0;
                        i / k; // raises exception

// resume comes back here, skipping the above statement

                        output << "Succeeded in nested layer.\n";

                  handler

                        case _EXCEPTION_DivisionByZero:

                              output <<
            "Caught _EXCEPTION_DivisionByZero in nested layer\n";

// We go back and resume at statement after the one that raised
// exception, i.e. after the statement "i / k".

                              resume;

                  endlayer;

                  output << "Caught _EXCEPTION_DivisionByZero\n";

// Repair cause of exception and repeat the call to globalFun()
// in the outer layer.

                  j = 5;
                  repeat;

      endlayer;

      output << "Good-bye World!\n";
end;
```

The output of the example is as follows.

```
Hello World!
```

```
Starting layer.
Caught _EXCEPTION_DivisionByZero in nested layer
Succeeded in nested layer.
Caught _EXCEPTION_DivisionByZero
ret is: 47
Good-bye World!
```

Chapter VIII.7. Shallow and Deep Exceptions

System exceptions are implicitly raised by Z47 at run time. The compiler has no way of knowing whether or not these exceptions will be raised. On the other hand, the compiler is aware of user exceptions.

In Z++ user exceptions can be used in two different ways. One can **explicitly** raise a user exception, or instruct the compiler to raise the exception **implicitly**. A user exception that is explicitly raised, via the **raise** statement, is called a **shallow** exception.

With regard to invariants, we instruct the compiler to **implicitly raise** an exception when the conditions of invariants are violated. A **deep** exception is a user exception raised **implicitly by the compiler**.

The compiler keeps track of user exceptions with respect to handlers. For each shallow exception appearing in a **raise** statement the compiler will look for a handler. In the following example the compiler reports a warning that we explain right after the example.

```
// Sample.zpp

#include<exception.h>
using namespace exceptionSpace;

// ------------------------------------------------------------- //

enum userExceptionEvents : exceptionEventType {
      _EXCEPTION_FirstUserException,
      _EXCEPTION_SecondUserException,
      _EXCEPTION_ThirdUserException
};

// ------------------------------------------------------------- //

void funLevelTwo(void) throws(_EXCEPTION_FirstUserException)
      raise(_EXCEPTION_FirstUserException);
end;

// ------------------------------------------------------------- //

void funLevelOne(void)
      funLevelTwo();
end;

////////////////////////////////////////////////////////////////////
```

```
entry void main(void)

      funLevelOne();

end;
```

The compiler reports the following **warning**.

Warning 59950: (Exception Tracer) At end of entry point 'main' following shallow exception(s) remain unhandled: _FirstUserException

We know that a Z++ program can only be **entered** from an **entry** point. Therefore, the compiler reports unhandled exceptions at **entry** points.

The way to look at the report is like this. For each call in an **entry** point, to a global function or a method, the compiler traces the nested calls. It collects the exceptions explicitly raised, and on its way back it looks for handlers. When it reaches the **entry** point, it reports all exceptions for which it **did not find a handler**.

The lack of a handler for a shallow exception is only reported as warning. This is because when a Z++ program is used as a **module**, exceptions are passed on to the program that invoked the **entry** point of the module.

The way compiler locates unhandled **deep** exceptions is analogous. However, unhandled deep exceptions are reported as **error**, at **entry** points. This is because deep exceptions must be handled within the program, and makes no sense to pass them on to the program that invoked the **entry** point. For instance, an object that violates its invariants is used **internally** within a module, and the user of the module has no access to the object.

Remark. Unless exception tracing is turned on, the error for deep exceptions will not be caught. Exception tracing is automatically turned on for Debug mode, and off for Release mode (when using IDE).

Chapter VIII.8. Throws specification

The **throws** specification is readability help with regard to **shallow** exceptions. A function or method cannot **explicitly** raise exceptions that are not listed in its **throws** specification. In particular for a function with a large body, it is not easy to see all the exceptions explicitly raised in its definition. The **throws** list is a simply way of documenting all the shallow exceptions, explicitly raised, in the body of a function.

The type mechanism takes the **throws** specification into account. That is, two functions with different **throws** lists are considered to be of different types.

In the example, the functions do not raise exceptions, yet they have **throws** list. The example illustrates the use of **type**. In general, it is better to keep the **throws** list consistent with the **actual** exceptions **explicitly** raised in the function. The compiler only complains when a function raises an exception not listed in its **throws** specification.

```
// Sample.zpp

#include<exception.h>
using namespace exceptionSpace;

// ------------------------------------------------------------------ //

enum userExceptionEvents : exceptionEventType {
      _EXCEPTION_FirstUserException,
      _EXCEPTION_SecondUserException,
      _EXCEPTION_ThirdUserException
};

// ------------------------------------------------------------------ //

type void singleFunType(void) throws(_EXCEPTION_FirstUserException);

type void doubleFunType(void)
            throws(_EXCEPTION_SecondUserException,
                   _EXCEPTION_ThirdUserException);

// ------------------------------------------------------------------ //

void singleFun(void) throws(_EXCEPTION_FirstUserException)
end;

void doubleFun(void) throws(_EXCEPTION_SecondUserException,
                            _EXCEPTION_ThirdUserException)
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
```

```
      singleFunType sftInstance = singleFun;
      doubleFunType dftInstance = doubleFun;

      sftInstance = dftInstance; // Error: type mismatch

end;
```

The last assignment in the above example is reported as **error**. That is because the object on the left `sftInstance` is of type `singleFunType`, but the object `dftInstance` on the right is of type `doubleFunType`. The **difference** between the two types is only in their **throws** specification.

Chapter VIII.9. Propagation of Exceptions

An exception raised in a **task** or global **thread** is passed on to its parent. Similarly, an exception raised in a component is passed on the component that loaded it. This continues until either the exception is handled, or execution reaches the startup thread. Once we reach the final thread without handling the exception, Z47 will terminate the process.

The entity in which exception is raised (**task**, **component**) is **not destroyed**. The following example shows that **task** object **persists** after exception being raised in one of its methods, and caught in its parent thread.

The same thing happens with regard to a loaded component. Only the **entry** point of the component that caused the exception ends. The parent (the one that called the **entry** point of component) receives the exception `_EXCEPTION_ComponentRaisedException`. The component is not unloaded.

```
// Example.zpp

#include<iostream.h>
#include<exception.h>

using namespace ioSpace;
using namespace exceptionSpace;

// ------------------------------------------------------------------ //
// Extend system exceptions

enum userExceptionEvents : exceptionEventType {
      _EXCEPTION_FirstUserException,
      _EXCEPTION_SecondUserException
};

// ------------------------------------------------------------------ //

task exceptionThreadType
      int counter;
public:
```

```
      exceptionThreadType(void);
      void UserEventGenerator(int);
end;

// ------------------------------------------------------------- //

exceptionThreadType::exceptionThreadType(void)
      counter = 0;
end;

void exceptionThreadType::UserEventGenerator(int i)
            throws(_EXCEPTION_SecondUserException)

      output << "Entering UserEventGenerator()\n";

// Next line may raise exception

      if (i < 47) raise _EXCEPTION_SecondUserException; endif;

// Next line will not print if exception is raised

      output << "Leaving UserEventGenerator()\n";
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      exceptionThreadType ETT;

      layer<userExceptionEvents>

// Next line will raise exception in thread of task object ETT.

            ETT.UserEventGenerator(7);

// Next line will not execute

            output << "Layer completed.\n";

      handler

case _EXCEPTION_FirstUserException .. _EXCEPTION_SecondUserException:

                  output << "Caught User-Defined Exception.\n";

      endlayer;

      output << "Outside of the exception layer.\n";

// Task object ETT is not destroyed for raising exception

      ETT.UserEventGenerator(97);
```

```
      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Entering UserEventGenerator()
Caught User-Defined Exception.
Outside of the exception layer.
Entering UserEventGenerator()
Leaving UserEventGenerator()
Good-bye World!
```

Chapter VIII.10. Branching of Exceptions

You can either linearly extend your exceptions or branch at any point. Consider the following two branches off of the system exceptions.

```
enum myFirstExceptions : exceptionEventType {
      _EXCEPTION_FirstUserException,
      _EXCEPTION_SecondUserException
};

enum mySecondExceptions : exceptionEventType {
      _EXCEPTION_FirstUserException,
      _EXCEPTION_SecondUserException
};
```

That is, instead of `mySecondExceptions` being an extension of `myFirstExceptions` you have decided to have both of your exceptions as extensions of the system exceptions. In particular, you are using the same literals for both exceptions. There is no difference between this branching, and linear extension in execution. The only difference is that if you use just the literal `_EXCEPTION_FirstUserException` the compiler will ask you to specify the type. That means, you need to use one of the following.

```
myFirstExceptions::_EXCEPTION_FirstUserException
mySecondExceptions::_EXCEPTION_FirstUserException
```

Other than that, there is no semantic difference between branching and linear extension.

Chapter IX. Conversions

Conversion, as defined in this chapter, applies to passing arguments to a call.

In an assignment, x = y, the y is argument passed to the operator =, invoked on object x. Therefore, conversions may be applied to y. As we shall illustrate, conversions apply to operands of other operators, as well.

Conversions take two forms with regard to Compiler's action: **implicit** and **explicit**. **Coercion** is an implicit conversion of numeric types. Either a smaller numeric type, say **int**, is coerced to a larger numeric type like **double**, or vice versa. In the latter case compiler generates warning about loss of precision.

For type constructors that admit methods (**class**, **task**, **collection** etc.) it is possible to define conversion from type being defined to a previously defined type. This is called a **reduction**. On the other hand, a **promotion** is a constructor for the type that accepts an instance of a previously defined type and constructs an instance of itself.

For fundamental types conversions only apply to pass by value. However, in other cases **conversions are applied even in pass by reference**. The purpose of pass by reference is not necessarily to modify the object passed to the call. Z++ compiler after performing the conversion passes a reference to the converted object, to the call. Thus if conversion takes place, **changes to the argument passed to the call will be lost**, as in pass by value.

Chapter IX.1. Conversions for Fundamental types

The conversion operator |− provides a convenient way of converting numeric types to string, and the other way round.

Chapter IX.2. Conversions for Type Constructors

A conversion is a mapping between two types. It converts an instance of one type to an instance of another type. The copy constructor is a mapping from a type to itself.

As mentioned in the introduction, a reduction is a conversion from type being defined to a previously defined type. A reduction is defined via a conversion operator returning an instance of the previously defined type. On the other hand, a promotion is a conversion from a previously defined type to the type being defined. A promotion is defined via a constructor taking an instance of the previously defined type as argument.

A constructor for a type can take several arguments. The term conversion (promotion) is only used in connection with constructors that take a single argument. The special constructor that takes an instance of type itself is called the copy constructor.

Both kinds of conversions can be specified as **cast** requiring **explicit cast** for their invocation. Otherwise, without **cast** specification, the compiler will invoke them implicitly as needed.

In this example reduction and promotion conversions are illustrated.

```
// Simple Conversions

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// Calls using conversions are explained in the main() entry point.
//--------------------------------------------------------------------//

//--------------------------------------------------------------------//
// Values for CollectionArgument
//--------------------------------------------------------------------//

enum argumentEnum {_one, _two, _three};

//--------------------------------------------------------------------//

struct ArgumentOne
      int i;

      ArgumentOne(void);
      ArgumentOne(int); // promotion
end;

ArgumentOne::ArgumentOne(void)
      i = 97;
end;

ArgumentOne::ArgumentOne(int n)
      i = n;
```

```
end;

//-------------------------------------------------------------------//

struct ArgumentTwo
      double d;

      ArgumentTwo(void);
end;

ArgumentTwo::ArgumentTwo(void)
      d = 97.47;
end;

//-------------------------------------------------------------------//

struct ArgumentThree
      short s;

      ArgumentThree(void);
end;

ArgumentThree::ArgumentThree(void)
      s = 7;
end;

//-------------------------------------------------------------------//
// CollectionArgument
//-------------------------------------------------------------------//

collection CollectionArgument<argumentEnum> {
      _one<ArgumentOne>,
      _two<ArgumentTwo>,
      _three<ArgumentThree>

      operator ArgumentTwo(void); // conversion operator
};

//-------------------------------------------------------------------//

CollectionArgument::operator ArgumentTwo(void)
      return self[_two];
end;

////////////////////////////////////////////////////////////////////
//-------------------------------------------------------------------//

struct largeType
      ArgumentOne one;
      double d;

      largeType(void);
      largeType(const ArgumentOne&, double);
      operator ArgumentOne&(void);
end;

//-------------------------------------------------------------------//
```

```
largeType::largeType(void)
      d = 11.77;
end;

largeType::largeType(const ArgumentOne& n, double b) : one(n), d(b)
end;

largeType::operator ArgumentOne&(void)
      return one;
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//

void globalFunOne(ArgumentTwo t)
      output << "\nIn globalFunOne...\n";
      output << "ArgumentTwo.d is: " << t.d << '\n';
end;

//--------------------------------------------------------------------//

void globalFunTwo(ArgumentOne n)
      output << "\nIn globalFunTwo...\n";
      output << "ArgumentOne.i is: " << n.i << '\n';
end;

//--------------------------------------------------------------------//

void globalFunThree(largeType lt)
      output << "\nIn globalFunThree...\n";
      output << "largeType.ArgumentOne.i is: " << lt.one.i << '\n';
      output << "largeType.d is: " << lt.d << '\n';
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// We pass instance of collection to the call. The argument ca is
// reduced to an instance of ArgumentTwo via conversion operator.

      CollectionArgument ca;
      globalFunOne(ca);

// The argument lt is reduced to an instance of ArgumentOne
// via conversion operator.

      largeType lt;
      globalFunTwo(lt);

// ArgumentOne(777) promotes the integer 777 to an instance of
// ArgumentOne. Then, the constructor of largeType takes the
```

```
// two arguments and constructs an instance of largeType.

      globalFunThree(largeType(ArgumentOne(777), 33.27));

      output << "Good-bye World!\n";
end;
```

The output of this program is as follows.

```
Hello World!

In globalFunOne...
ArgumentTwo.d is: 97.47

In globalFunTwo...
ArgumentOne.i is: 97

In globalFunThree...
largeType.ArgumentOne.i is: 777
largeType.d is: 33.27
Good-bye World!
```

Chapter IX.3. Array Conversions

Z++ does not hinder your freedom of speech. Passing an array by value is expensive. Nonetheless, the language should not make that decision for you.

Suppose you need to convert an array to a different type so you can pass a reference to the converted array, to the call. You will need to do the conversion manually, and then pass the result by reference. In such a situation, when passing an array by value Z++ does exactly the same.

Array conversions apply to regular and dynamic arrays.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////

struct firstCellType
      int i;

      firstCellType(void);
      ~firstCellType(void);
      firstCellType(int); // promotion
      operator int (void); // reduction
end;

firstCellType::firstCellType(void)
      i = 47;
end;

firstCellType::~firstCellType(void)
      output << "\nIn firstCellType::~firstCellType(void).\n\n";
end;

firstCellType::firstCellType(int n)
      i = n;
end;

firstCellType::operator int (void)
      return i;
end;

// ---------------------------------------------------------------- //

struct secondCellType
      firstCellType fct;

      ~secondCellType(void);
      secondCellType(const firstCellType&); // promotion
      operator firstCellType& (void); // reduction
end;
```

```
secondCellType::~secondCellType(void)
      output << "\nIn secondCellType::~secondCellType(void).\n\n";
end;

secondCellType::secondCellType(const firstCellType& e) : fct(e)
end;

secondCellType::operator firstCellType& (void)
      return fct;
end;

////////////////////////////////////////////////////////////////////////

typedef firstArrayType firstCellType[3];

// ------------------------------------------------------------------ //
// We pass an array of secondCellType to it. This will invoke
// secondCellType::operator firstCellType& (void).
// ------------------------------------------------------------------ //

void globalFirst(firstArrayType fat)
  output << "\nIn globalFirst().\n";

// In output, fat[i] via conversion returns fat[i].i

  for (int i = 0; i < 3; i++)
    output << "fat[i] at index: " << i << " is: " << fat[i] << '\n';
  endfor;
end;

// ------------------------------------------------------------------ //
// Passing array of firstCellType,
// invoking secondCellType(const firstCellType&).
// ------------------------------------------------------------------ //

typedef secondArrayType secondCellType[3];

void globalSecond(secondArrayType sat)
  output << "\nIn globalSecond().\n";

// In output, fct via conversion returns fct.i

  for (int i = 0; i < 3; i++)
    output << "sat[i].fct at index: " << i << " is: " <<
       sat[i].fct << '\n';
  endfor;
end;

////////////////////////////////////////////////////////////////////////
// Reduction using firstCellType::operator int (void).
// ------------------------------------------------------------------ //

typedef intArray int[3];

void globalFirstInt(intArray ia)
  output << "\nIn globalFirstInt().\n";
```

```
  for (int i = 0; i < 3; i++)
    output << "ia[i] at index: " << i << " is: " << ia[i] << '\n';
  endfor;
end;

// ------------------------------------------------------------- //
// Promotion using firstCellType::firstCellType(int n).
// ------------------------------------------------------------- //

void globalSecondInt(firstArrayType fat)
  output << "\nIn globalSecondInt().\n";

// fat[i] via conversion returns fat[i].i

  for (int i = 0; i < 3; i++)
    output << "fat[i] at index: " << i << " is: " << fat[i] << '\n';
  endfor;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// ------------------------------------------------------------- //
// Reduction from secondCellType to firstCellType.

      secondCellType asct[3];
      for (int i = 0; i < 3; i++) asct[i].fct.i += (i + 5); endfor;
      globalFirst(asct);

// ------------------------------------------------------------- //
// Promotion from firstCellType to secondCellType

      firstCellType afct[3](97);
      for (int i = 0; i < 3; i++) afct[i].i += (i * 7); endfor;
      globalSecond(afct);

// ------------------------------------------------------------- //
// Reduction from firstCellType to int.

      for (int i = 0; i < 3; i++) afct[i].i += (i - 7); endfor;
      globalFirstInt(afct);

// ------------------------------------------------------------- //
// Promotion from int to firstCellType

      int arrayArg[3](777);
      globalSecondInt(arrayArg);

// ------------------------------------------------------------- //

      output << "Good-bye World!\n";
```

```
end;
```

The output of this program is as follows.

Note. We are only showing the output of global functions.

```
In globalFirst().
fat[i] at index: 0 is: 52
fat[i] at index: 1 is: 53
fat[i] at index: 2 is: 54

In globalSecond().
sat[i].fct at index: 0 is: 97
sat[i].fct at index: 1 is: 104
sat[i].fct at index: 2 is: 111

In globalFirstInt().
ia[i] at index: 0 is: 90
ia[i] at index: 1 is: 98
ia[i] at index: 2 is: 106

In globalSecondInt().
fat[i] at index: 0 is: 777
fat[i] at index: 1 is: 777
fat[i] at index: 2 is: 777
```

Chapter IX.4. Conversions for Symmetric operators

Logical operators are symmetric in that the result of their evaluation is independent of the order of their operands. For instance, `(x && y)` is identical to `(y && x)`.

For logical operators Z++ compiler will first apply available conversions to **boolean** before trying operator overloading.

For symmetric operators conversions, when available, are applied to both operands. Thus, in evaluating `(x && y)` compiler will first try to convert both, x and y to **boolean**. If left operand x is not **boolean**, and does not provide conversion to **boolean**, only then the compiler will try overloading with respect to left operand.

In the next example we are mixing overloading and conversions. This is only done to show how compiler handles such complex expressions.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////
// Type convertible can reduce itself to boolean.
// It promotion from boolean is only used for construction.
// ---------------------------------------------------------------- //

struct convertible
      boolean b;

      convertible(void);
      convertible(boolean);
      operator boolean (void);
end;

convertible::convertible(void)
      b = True;
end;

convertible::convertible(boolean bb)
      b = bb;
end;

convertible::operator boolean (void)
      return b;
end;

// ---------------------------------------------------------------- //
// Type convertibleOverload uses operator && to convert
// an instance of int to an instance of convertible.

struct convertibleOverload
      convertible c;
```

```
      convertibleOverload(boolean);
      convertible& operator&&(int);
end;

convertibleOverload::convertibleOverload(boolean b) : c(b)
end;

convertible& convertibleOverload::operator&&(int i)
      return c;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// ------------------------------------------------------------------ //
// (co && coArgument) invokes overloaded && returning a
// reference to an instance of convertible. Note that
// co does not provide conversion to boolean so operator
// overloading is applied.
// The returned object reduces itself to a boolean.
// cvb also reduces itself to an instance of boolean.
// Since both operands are true, the second logical
// operator && evaluates to true.

      int coArgument = 7;
      convertibleOverload co(True);
      convertible cvb;

      if (co && coArgument && cvb)
            output << "(co && coArgument && cvb) is true.\n";
      endif;

// ------------------------------------------------------------------ //

      output << "\nGood-bye World!\n";

end;
```

The output of this program is as follows.

```
Hello World!
(co && coArgument && cvb) is true.

Good-bye World!
```

Chapter IX.5. Conversions for Asymmetric operators

An operator is asymmetric when the order of its operands is significant. For instance, the expression `(x < y)` is not identical to `(y < x)`. In this case the compiler will first try operator overloading, and then attempt to look for a conversion for the operand on the right hand side.

Operators `==` and `!=` are treated like other comparison operators even though they are symmetric operators. That is, first overloading is checked, and then conversions are searched for the right hand side operand.

Chapter X. Operators

Symbolic operators are international and easier to use than words. Furthermore, the use of operators as opposed to function calls makes expressions more readable.

Here is a brief review of some terms used in this chapter.

Reduction, Promotion and Coercion

A conversion of an operand means either a **reduction** or a **promotion**. Conversions that are the result of applying a conversion operator reduce the type of the operand. On the other hand, conversions that result from an application of a constructor promote the type of the operand. In case of numerical conversions, a reduction is a conversion to a smaller type, while promotion is a conversion to a larger type. For instance, converting an **int** to a **short** is a reduction, while the opposite is a promotion.

Converting a numeric type to another numeric type is called **coercion**. In particular, a reduction generally involves loss of precision. For instance coercing an object of type **double** to an object of type **int** could result in loss of precision.

Structural Types

A **type constructor** is a mechanism for defining new types. When the mechanism allows attaching a function to the type it defines, we refer to the function as the **method** of the type. A method can only be invoked on instances of its type. Types defined via type constructors that admit methods are called **structural types**. Structural types in Z++ are defined via the mechanisms **struct**, **class**, **task**, **frame**, **union**, **collection** and **bitfields**.

Compatible Arrays

Two arrays are compatible when they have the same number of dimensions, and the sizes of corresponding dimensions are equal. This definition applies to dynamic and static arrays. In general, however, the compiler is unable to verify the compatibility of two dynamic arrays.

Compiler knows the sizes of dimensions of static arrays and is therefore able to verify the compatibility of two static arrays. Consider the following.

```
int firstArray[3][4];
int secondArray[3][4];
```

In above example the two static arrays are compatible, which can be determined by the user and the compiler alike. Now, consider dynamic arrays.

```
int dimOne = 3, dimTwo = 4;
```

```
int firstArray[dimOne][dimTwo];
int secondArray[dimOne][dimTwo];
```

In the above example the compiler will consider the two dynamic arrays compatible. It makes its determination based on the objects used for the sizes of dimensions. However, if one of those objects receives a different value prior to declaring the second array, the conclusion will be wrong, causing the raising of an exception during execution.

Some operators such as the assignment operators do not support dynamic array operands and some like the comparison operators do support them. When dynamic arrays are not directly supported, one has to use a loop. That way, at run time one can determine whether dynamic arrays are compatible before entering the loop. Note that the number of dimensions is known at compile time.

```
int firstSize = size(firstArray, 0); // size of first dimension
int secondSize = size(firstArray, 1); // size of second dimension

if (firstSize == size(secondArray, 0) &&
            secondSize == size(secondArray, 1))

      for (int first = 0; first < firstSize; first++)
         for (int second = 0; second < secondSize; second++)

            firstArray[first][second] = secondArray[first][second];

         endfor;
      endfor;
endif;
```

Chapter X.1. Assignment Operators

Assignment operators are **binary**. The result of operation between left and right operands becomes the new literal value for the **left operand**.

Assignment operators do not accept dynamic arrays except in specifically indicated cases.

The following is the list of assignment operators. The overloading of operators is checked **before** their standard meaning, indicated in the following list, are considered.

= assignment. Make left operand same as right operand.
+= sum. Make left the sum of left and right.
-= difference. Make left the difference (left – right).
*= product. Make left the product of left and right.
/= division. Make left the ratio (left / right).
%= remainder. When dividing whole numbers, make left the remainder of division.
*^= power. Make left the result of raising left to the power of right.
~= bit inversion. Invert bits of right operand and assign it to left operand.
&= arithmetic and. Left becomes the result of bit-wise and of left and right.
|= arithmetic or. Left becomes the result of bit-wise or of left and right.
^= arithmetic exclusive or. Left becomes the result of exclusive or of left and right.
<<= shift left. Shift left operand to the left by size of right operand then assign it to left.
>>= shift right. Shift left operand to the right by size of right operand then assign it to left.

The unary bit inversion operator ~ does not change its operand. Instead it returns a temporary for the result. The binary form in this chapter can be used as x ~= x where both operands are the same. This will invert the bits of object x and assign it to itself.

Remark. Suppose X and Y are arrays of **class** C, and operator += is overloaded for the **class** C. The array expression X += Y will return the array X. That is, an **overloaded** array operation will always return the array on which an operator is called. This is in part because the return of the overloaded operator could be **void**.

Section X.1.A. Numeric Assignment

When types of left and right operands do not match, the right operand is converted to a numeric object of the type of the left operand before making the assignment.

When coercion is a reduction, that is the right operand is converted to a smaller type, the compiler generates a warning with regard to loss of precision. Casting the right operand to the type of left operand, as shown below, avoids the warning.

```
double rightOperand = 2.5;
int leftOperand = int(rightOperand); // leftOperand == 2
```

The compiler also reports a warning when one operand is unsigned and the other operand is signed.

When right operand is of structural type, the compiler will look for implicit conversions to the type of left operand. When the right operand provides multiple implicit conversions to numeric types none of which matches the type of left operand, compiler selects the conversion with largest numeric type and generates a warning.

Section X.1.B. Bitfields Assignment

The example of chapter Bitfields illustrates the type constructor **bitfields**. Since operator overloading extends to arrays, here we show an array example, using regular (static) arrays, as well as dynamic arrays.

Operator overloading is checked before making assignment. So, in following lines from the example, overloaded operators are invoked on the dynamic arrays, rather than making assignments.

```
dynFlagsArray &= argument;
dynFlagsArray ^= argument;
dynFlagsArray += argument;
dynFlagsArray *= argument;
```

Here is the example.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////

bitfields flags
      first 3;
      second 5;
      third 7;
```

```
// Methods

	flags(void);
	uint badMethod(uint);

	uint operator&=(uint);
	uint operator^=(uint);
	uint operator+=(uint);
	uint operator*=(uint);
end;

// ------------------------------------------------------------------ //

flags::flags(void)
	first = uint(5); // binary 101
	second = uint(3); // binary 00011
	third = uint(16); // binary 0010000
end;

// Ending numeric value for first is 17, so it becomes 1.

uint flags::badMethod(uint u)
	first = (third += (second &= first));
	return third;
end;

// ------------------------------------------------------------------ //

uint flags::operator&=(uint u)
	return first &= u;
end;

uint flags::operator^=(uint u)
	return second ^= u;
end;

uint flags::operator+=(uint u)
	return second += u;
end;

uint flags::operator*=(uint u)
	return third *= u;
end;


////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

	output << "Hello World!\n";

	flags f;

	flags flagsArray[3];
```

```
      flagsArray.badMethod(2);

      for (int index = 0; index < 3; index++)
            output << "\nflagsArray[index].first is: " <<
                    flagsArray[index].first << '\n';
            output << "flagsArray[index].second is: " <<
                    flagsArray[index].second << '\n';
            output << "flagsArray[index].third is: " <<
                    flagsArray[index].third << '\n';
      endfor;

// ------------------------------------------------------------------ //

      int dimOne = 3;
      int dimTwo = 4;

      flags dynFlagsArray[dimOne][dimTwo];

      uint argument = 7;

      dynFlagsArray &= argument;
      dynFlagsArray ^= argument;
      dynFlagsArray += argument;
      dynFlagsArray *= argument;

      for (int One = 0; One < dimOne; One++)
            for (int Two = 0; Two < dimTwo; Two++)

                  output << "\ndynFlagsArray.first is: " <<
                          dynFlagsArray[One][Two].first << '\n';
                  output << "dynFlagsArray.second is: " <<
                          dynFlagsArray[One][Two].second << '\n';
                  output << "dynFlagsArray.third is: " <<
                          dynFlagsArray[One][Two].third << '\n';

            endfor;
      endfor;

      output << "\nGood-bye World!\n";

end;
```

The output is shown below.

```
Hello World!

flagsArray[index].first is: 1
flagsArray[index].second is: 1
flagsArray[index].third is: 17

flagsArray[index].first is: 1
flagsArray[index].second is: 1
flagsArray[index].third is: 17

flagsArray[index].first is: 1
flagsArray[index].second is: 1
```

```
flagsArray[index].third is: 17

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

dynFlagsArray.first is: 5
dynFlagsArray.second is: 11
dynFlagsArray.third is: 112

Good-bye World!
```

Section X.1.C. Function Assignment

A function pointer is defined via **type**. A function, or function pointer can be assigned to a function pointer only when all specifications, such as type of return and signature, threading, signals etc. match exactly. For examples see Function Pointer.

Section X.1.D. Array Assignment

All assignment-related operators take array operands. Both operands can be arrays, or only left operand can be array.

When both operands are arrays, the kind of arrays must be static. It is not possible to determine the size of dynamic arrays during compile time. Note that a static array means that the sizes of its dimensions are known during compile time. The term static is not used in Z++.

Static array operands must be compatible. They must have the same number of dimensions, and sizes of corresponding dimensions must be the same. For instance the two static arrays below are compatible.

```
int intArray[3][5];
double doubleArray[3][5];
doubleArray += intArray;
```

For numeric arrays all operators are overloaded with their standard semantics. In addition, for arrays of structural types the overloading of operators extends to the entire array. In particular arrays of structural types can be assigned because the compiler automatically overloads the operator assign = for structural types.

When only left operand is array the kind of array can also be dynamic. Either static or dynamic array, the operator is applied to all cells of array on the left using the same right operand. The following shows a two-dimensional dynamic array of numeric type on the left.

```
int first = 4, second = 7;
double doubleArray[first][second];
doubleArray += 47.97;
```

Compiler looks for possible implicit conversions for array operands as it does otherwise.

Examples are in chapter Array Conversions.

Section X.1.E. Special Assignments

When the indicated operator is invalid for the types of operands, the compiler reports that as an error. For instance, the operator += with left operand of type pointer, can only take a discrete numeric for its right operand.

However, Z++ strings involve special forms of assignment. One operand can be (one-dimensional) static array of characters and the other string. When the left operand is array of characters the compiler generates a warning because the string may be too long for the size of array.

The operator = can also take one operand of type pointer to character, and the other string. However, in this case, when pointer is the left operand one of two things can happen. In a declaration the compiler allocates space and generates a warning that the space must be freed manually. In an assignment the compiler does not allocate space but generates a warning to do so. Below are some examples.

```
char* p = "Hello there."; // space allocated automatically
string s = "Good morning.";
char* q;
q = s; // will crash at runtime, first allocate space
q = p; // OK, plain pointer assignment
```

Section X.1.F. Enumeration Assignments

Enumeration types are extensible. However, the lack of methods for enumeration type makes the use of polymorphism ineffective. Instead, objects of enumeration type can be assigned to objects of types extended from a base, as illustrated below.

```
// Type definitions

enum primes {_three = 3, _five = 5, _seven = 7, _eleven = 11};
enum morePrimes : primes {_thirteen = 13, _seventeen = 17};

// Declarations and execution

primes pm = _five;
morePrimes mpm = pm;

output << [mpm]; // prints 5
```

However, assignment of an object for an extended type to an object of base type does not work, and the compiler reports error. This is because a derived type includes the literals of all of its bases, while an object of a base type cannot assume a literal of a derived type.

Chapter X.2. Arithmetic Operators

Arithmetic operators are **binary**. These operators can take static array arguments for the left operand, or both operands. Arithmetic operators do **not change** their operands. Instead they return a temporary for the result of the operation.

For **numeric** arrays, these operators are **overloaded** with same semantics as for the numeric type of the cells of arrays.

Compiler checks the overloading of arithmetic operators before attempting to apply conversions supplied by user. Type checking and conversion rules for these operators are similar to their corresponding Assignment Operators.

`+` plus. Add two numbers.
`-` minus. Subtract the right operand from the left operand.
`*` multiply. Multiply two numbers.
`/` divide. Divide the left operand by the right operand.
`%` remainder. Remainder of dividing two whole numbers.
`*^` power. Raise the left operand to the power of right operand.
`&` arithmetic and. Bit-wise **and** the two operands.
`|` arithmetic or. Bit-wise **or** the two operands.
`^` arithmetic exclusive or. Bit-wise **exclusive or** the two operands.
`<<` shift left. Shift left the left operand by the size of right operand.
`>>` shift right. Shift right the left operand by the size of right operand.

Remark. Suppose X and Y are arrays of **class** C, and operator + is overloaded for the **class** C. The array expression X + Y will return the array X. That is, an **overloaded** array operation will always return the array on which an operator is called. This is in part because the return of the overloaded operator could be **void**.

Chapter X.3. Comparison Operators

Comparison (or relational) operators are as follows.

`<` means **less**. The condition `(x < y)` is true when x is less than y.
`<=` means **less or equal**. `(x <= y)` is true when x equals y, or x is less than y.
`>` means **more**. The condition `(x > y)` is true when x is more than y.
`>=` means **more or equal**. `(x >= y)` is true when x equals y, or x is more than y
`==` means **equal**. `(x == y)` is true when object x is equal to object y.
`!=` means **not equal**. `(x != y)` is true when x and y are not equal.

Comparison operators take **dynamic** and **static arrays** as operands, as well. In this chapter we discuss various **implicit conversion** the compiler does while evaluating relational expressions. Furthermore, comparison operators can be **chained**.

Remark. Suppose X and Y are arrays of **class** C, and operator `<=` is overloaded for the **class** C. The array expression X `<=` Y will return the array X. That is, an **overloaded** array operation will always return the array on which an operator is called. This is in part because the return of the overloaded operator could be **void**. In particular, remember that operators `==` and `!=` are always overloaded for structural types.

Section X.3.A. Chaining Comparison Operators

Consider the following expression, which may appear as condition to an if-clause, while-loop, and so on.

```
(x < y && y < z)
```

Chaining means that the above expression can be written using only relational operators, and without logical operators, as shown below.

```
(x < y < z)
```

A relational chain can contain as many operators as you need. The only restriction is that a chain can only contain operators `<` and `<=` or else `>` and `>=`. Operators `!=` and `==` can be used in all combinations. Below are some examples of chaining. Note that, the logical operator `&&` (short and) is assumed for chaining.

`(x < y <= z == u < v)` means `(x<y && y<=z && z==u && u<v)`
`(a >= b >= c > d != e)` means `(a>=b && b>=c && c>d && d!=e)`

Sometimes it is convenient to instruct the compiler to use operator `||` instead of `&&`. This is accomplished by enclosing the chained relational expression between brackets.

`(![x < y < z])` means `(!(x<y || y<z))`

Note that it is the use of **brackets** that changes the semantics, as shown below where parentheses are used instead of brackets.

`(!(x < y < z))` means `(!(x<y && y<z))`

Section X.3.B. Overloading Comparison Operators

Overloading of comparison operators in a relational expression takes **priority**. That is, the overloading of an operator is checked **before** its built-in semantics are examined. This is particularly important in conjunction with chaining. Below is an illustration.

```
// A structure that converts int to boolean
struct intConvertor
      boolean operator<=(int);
end;

boolean intConvertor::operator<=(int n)
      return n > 0 ? True : False;
end;

// A structure that converts a double to intConvertor
struct doubleConvertor
      intConvertor ic;

      intConvertor& operator<(double);
end;

intConvertor& doubleConvertor::operator<(double d)
      return ic;
end;

// Declarations

int Integer = 7;
doubleConvertor dc;

boolean bb = dc < 3.1 <= Integer;
```

In above example, the sub-expression `dc < 3.1` returns a reference to member ic transforming it to expression `ic <= Integer` . The latter returns a **boolean**, which is used to initialize bb. Thus in this case no chaining took place. The expression was evaluated in the usual way, using operator overloading.

Section X.3.C. String Comparisons

Strings can be compared to **character pointers** and to one-dimensional **arrays** of characters. Two character pointers are compared as pointers, **not as string**. The string comparison is applied when at least **one operand is of type string**. Similarly, two arrays are always compared **cell-wise**.

The example of this section illustrates several combinations of comparing, **string**, **char*** and structures with conversions to these types.

The example also illustrates priorities between conversions and chaining, and how to enforce one or the other.

The declaration of **char** pointers `leftPointer` and `localCharPtr` is commented as: Compiler warning. The warning says the following.

Warning 13052: Memory is automatically allocated for : 'localCharPtr' but must be deleted manually.

That is, for a declaration as shown below:

```
      char* localCharPtr = "Much"; // Compiler warning
```

the compiler allocates space. However, it has no clue when you will not need the space any more, so it generates a warning as a reminder. We are deleting the pointers at end of example.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //
// A structure with conversion to string

struct stringConvertor
      string s;

      stringConvertor(void);
      stringConvertor(string);
      operator string& (void);
end;

stringConvertor::stringConvertor(void)
      s = "XYZ";
end;

stringConvertor::stringConvertor(string g)
      s = g;
end;
```

```
stringConvertor::operator string& (void)
      return s;
end;

// ------------------------------------------------------------- //
// A structure with conversion to char*

struct charPtrConvertor
      string s;
      char a[5];

      charPtrConvertor(string);
      operator char* (void);
end;

charPtrConvertor::charPtrConvertor(string g)
      s = g;
      a = s;
end;

charPtrConvertor::operator char* (void)
      return (char*) &a;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// ------------------------------------------------------------- //
// When Left operand is string, Right operand can be string,
// char array or structure with conversion to string.

      string leftString = "ABC";
      char rightArray[4] = "DEF";
      stringConvertor sc;

      if (leftString < rightArray <= sc)
            output << "(leftString < rightArray <= sc) is True.\n";
      endif;

// ------------------------------------------------------------- //
// Both sides are structural.

      stringConvertor scLeft("abcd");
      stringConvertor scMiddle("efg");
      stringConvertor scRight("hijkl");
      stringConvertor scLast("Last one.");

  if (scLeft <= scMiddle < scRight != scLast)
    output << "(scLeft <= scMiddle < scRight != scLast) is True.\n";
  endif;
```

```
// If you need to use scMiddle for two different conversions,
// use the usual syntax. In a chain, once scMiddle is
// converted to string, that will be used for next operator.

  if (scLeft <= scMiddle && scMiddle < scRight != scLast)
   output <<
     "(scLeft <= scMiddle && scMiddle < scRight != scLast) is True.\n";
  endif;

// ------------------------------------------------------------------ //
// Both sides are structural.

      charPtrConvertor cpcLeft("abcd");
      charPtrConvertor cpcRight("xyz");

// cpcLeft needs to be converted twice, so we use usual syntax.

  if (scLeft <= cpcLeft && cpcLeft < scMiddle < cpcRight)
   output <<
     "(scLeft <= cpcLeft && cpcLeft < scMiddle < cpcRight) is True.\n";
  endif;

      char* leftPointer = "uvxyz"; // Compiler warning

      if (leftPointer > scLeft.s) // right is string member s
            output << "(leftPointer > scLeft.s) is True.\n";
      endif;

      if (leftPointer > scLeft) // right is structure
            output << "(leftPointer > scLeft) is True.\n";
      endif;

// Right operand is string. Left operand can convert to string/char*

      string localStg = "Less";
      stringConvertor scTwoLeft("More");
      charPtrConvertor cpcTwoLeft("Less");

      if (scTwoLeft > localStg) // string conversion
            output << "(scTwoLeft > localStg) is True.\n";
      endif;

      if (cpcTwoLeft == localStg) //char* conversion, string comparison
            output << "(cpcTwoLeft == localStg) is True.\n";
      endif;

// Right operand is char*. Left operand can reduce to string.

      char* localCharPtr = "Much"; // Compiler warning

      if (scTwoLeft <= localCharPtr)
            output << "(scTwoLeft <= localCharPtr) is True.\n";
      endif;

// Compiler generates warning for deleting the following
// pointers. Memory is allocated, but user must release it.
```

```
      delete leftPointer;
      delete localCharPtr;

// ------------------------------------------------------------------ //

      output << "\nGood-bye World!\n";

end;
```

The output of the example is as follows.

```
Hello World!
(leftString < rightArray <= sc) is True.
(scLeft <= scMiddle < scRight != scLast) is True.
(scLeft <= scMiddle && scMiddle < scRight != scLast) is True.
(scLeft <= cpcLeft && cpcLeft < scMiddle < cpcRight) is True.
(leftPointer > scLeft.s) is True.
(leftPointer > scLeft) is True.
(scTwoLeft > localStg) is True.
(cpcTwoLeft == localStg) is True.
(scTwoLeft <= localCharPtr) is True.

Good-bye World!
```

Section X.3.D. Conversions in Chains

Conversions and chaining was illustrated in the example of previous section. Here we give another illustration.

The important thing to observe is that, when an object is converted it is used as operand for next operator in a chain. Use the usual logical operators at the point that you do not want the compiler to do its predetermined routine for chaining.

Here is an illustration.

```
// Definition of type

struct twoConversions
      int i;
      string s;

      twoConversions(void);
      operator int (void);
      operator string& (void);
end;

twoConversions::twoConversions(void)
      i = 97;
      s = "Good-bye";
end;

twoConversions::operator int (void)
      return i;
end;

twoConversions::operator string& (void)
      return s;
end;

// Declarations

twoConvertions tc;
string stg = "Hi There.";
int Integer = 47;

// Usage

if (stg < tc <= Integer) endif; // Compiler reports error
if (stg < tc && tc <= Integer) endif; // No error
```

In above if-condition, stg is of type **string**. So, compiler converts the object tc to an object of type **string**. In first case this causes error because the converted object of type **string** cannot be compared to Integer. In second case, compiler converts tc once to **string**, and then to **int**. There is no error here.

Section X.3.E. Enumeration Comparisons

Objects of enumeration type can be compared, either when they are of the same type, or one object's type is an extension of the type of the other. Below is an illustration.

```
// Definitions

enum baseEnum {_one, _two};
enum derivedEnum : baseEnum {_three};

struct enumConvertor
      baseEnum be;

      enumConvertor(void);
      operator baseEnum (void);
end;

enumConvertor::enumConvertor(void)
      be = _one;
end;

enumConvertor::operator baseEnum (void)
      return be;
end;

// Declarations and usage

enumConvertor ec;
baseEnum be = _two;
derivedEnum de = _three;

if (be != de >= ec)
      output << "(be != de >= ec) is True.\n";
endif;
```

The output of this example is as follows.

```
(be != de >= ec) is True.
```

Section X.3.F. Pointer Comparisons

The usual C pointer comparison is supported in Z++. Two pointers can be compared using all the comparison operators.

Section X.3.G. Array Comparisons

Overloading of cells extends to arrays. The comparison operators are internally overloaded for numeric and string arrays. However, because array expressions always return the array on which an operator is invoked, for general arrays you will need to write a loop, as illustrated in the example of this section.

Comparison operators accept compatible **dynamic** and **static** arrays as operands. In particular, when the type of cells of an array provides a suitable implicit conversion, the compiler will apply the conversion to the entire array.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

struct intConvertor
	int i;

	intConvertor(void);
	operator int (void);
end;

intConvertor::intConvertor(void)
	i = 47;
end;

intConvertor::operator int (void)
	return i;
end;

// ------------------------------------------------------------------ //

struct doubleConvertor
	double d;

	doubleConvertor(void);
	doubleConvertor(double);
	operator double (void);
end;

doubleConvertor::doubleConvertor(void)
	d = 97.47;
end;
```

```
doubleConvertor::doubleConvertor(double b)
      d = b;
end;

doubleConvertor::operator double (void)
      return d;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// ------------------------------------------------------------------ //
// Numeric arrays do not need loop.

      int intArray[3](99);
      double doubleArray[3](99);

      if (intArray == doubleArray)
            output << "(intArray == doubleArray) is True.\n";
      endif;

// ------------------------------------------------------------------ //
// Static arrays of structures, with numeric conversion.

      intConvertor iclArray[3];
      intConvertor icrArray[3];
      icrArray[2].i = 117;

      int counter;
      boolean arrayEquality = False;

      for (counter = 0; counter < 3; counter++)
            if (iclArray[counter] != icrArray[counter])
                  arrayEquality = True;
                  break;
            endif;
      endfor;

      if (arrayEquality)
            output << "(iclArray != icrArray) is True.\n";
      endif;

// ------------------------------------------------------------------ //
// Static arrays of structures, with numeric conversion.

      doubleConvertor sLeftArray[3](4.7);
      doubleConvertor sRightArray[3](9.7);

      arrayEquality = True;

      for (counter = 0; counter < 3; counter++)
            if (!(sLeftArray[counter] < sRightArray[counter]))
```

```
                    arrayEquality = False;
                    break;
              endif;
        endfor;

        if (arrayEquality)
              output << "(sLeftArray < sRightArray).\n";
        endif;

// ------------------------------------------------------------------ //
// Dynamic arrays of structures, with numeric conversion.

  int dimOne = 3, dimTwo = 5;

  doubleConvertor dLeftArray[dimOne][dimTwo](4.7);
  doubleConvertor dRightArray[dimOne][dimTwo](9.7);

  arrayEquality = True;

  for (dimOne = 0; dimOne < size(dLeftArray, 0); dimOne++)
    for (dimTwo = 0; dimTwo < size(dLeftArray, 1); dimTwo++)

      if (dLeftArray[dimOne][dimTwo] > dRightArray[dimOne][dimTwo])
        arrayEquality = False;
        break;
      endif;

    endfor;
  endfor;

  if (arrayEquality)
    output << "(dLeftArray <= dRightArray).\n";
  endif;

// ------------------------------------------------------------------ //

      output << "\nGood-bye World!\n";

end;
```

Chapter X.4. Logical Operators

Logical operators are listed below.

`&&` short and: evaluates to false when at least one operand is false.
`||` short or: evaluates to true when at least one operand is true.
`<>` long and: evaluates to false when at least one operand is false.
`><` long or: evaluates to true when at least one operand is true.
`^^` exclusive or: evaluates to true exactly when one of its operands is true.
`!` negation: evaluate to true if its operand is false, and vice versa.

The short operators have the same semantics as in C. When the evaluation of an operand determines the value of the entire expression, the evaluation of the remaining operands is skipped. This is important in that the evaluation of operands that are skipped may invoke methods on objects, thereby changing their state.

In order to instruct the compiler to evaluate all operands before determining the value of the expression, use the long operators. Otherwise, the logical meaning of short and long operators is identical.

Note that in general the use of short and long operators are not equivalent. Long operators will always evaluate all their operands, which may result in method invocations on objects. Thus, though logically equivalent, the short and long operators are not identical.

Operators && and <> yields true exactly when both of their operands are true. On the other hand, operators || and >< yield true when at least one of their operands is true. The latter means that we will get true even when both operands are true. Sometimes we want the expression to yield true exactly when one of the operands is true. That is precisely what the exclusive or determines.

Note that the exclusive or always evaluates both of its operands, and is therefore a long logical operator. There is no short exclusive or.

Section X.4.A. Negation of Pointers and Arrays

For a pointer P, the negation is tested as (!P). The equivalent form (P != 0) is not supported. Similarly, the nullity of a pointer P is tested as (P), instead of (P == 0).

The same pattern applies to arrays. An array of pointers is null when at least one cell is null pointer. Stated differently, an array of pointers is not null only when none of its cells is null.

In the same vain, an array of **boolean** is true only when all cells are true. Below is an illustration.

```
//Nullity.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// ------------------------------------------------------------------ //
// Dynamic array of boolean.

      int dimOne = 3, dimTwo = 5;

      boolean booleanDArray[dimOne][dimTwo](True);

      if (booleanDArray)
            output << "booleanDArray is all true.\n";
      endif;

      booleanDArray[2][4] = False; // set last cell to false

      if (!booleanDArray)
            output << "booleanDArray is false.\n";
      endif;

      booleanDArray[2][4] = True;
      booleanDArray[0][0] = False; // make first cell false

      if (!booleanDArray)
            output << "booleanDArray is false again.\n";
      endif;

// ------------------------------------------------------------------ //
// Pointer arrays. Array is not NULL only if no cell is NULL.

      int* pointerDArray[dimOne][dimTwo](&dimOne);
```

```
      if (pointerDArray)
            output << "pointerDArray is not NULL.\n";
      endif;

      pointerDArray[1][1] = 0;

      if (!pointerDArray)
            output << "pointerDArray is NULL.\n";
      endif;

// ------------------------------------------------------------------ //

      output << "\nGood-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
booleanDArray is all true.
booleanDArray is false.
booleanDArray is false again.
pointerDArray is not NULL.
pointerDArray is NULL.

Good-bye World!
```

For an array of structures, if the type of cell provides a conversion to **boolean** the test for being true or false is done the same way, as illustrated below. The result of test is true only when all cells evaluate to true.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

struct booleanConvertor
      boolean b;

      booleanConvertor(void);
      operator boolean (void);
end;

booleanConvertor::booleanConvertor(void)
      b = True;
end;

booleanConvertor::operator boolean (void)
      return b;
end;
```

```
////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

	output << "Hello World!\n";

	booleanConvertor bmArray[3];

	if (bmArray)
		output << "Static bmArray is true.\n";
	endif;

	bmArray[2].b = False;
	if (!bmArray)
		output << "Static bmArray is false.\n";
	endif;

	output << "\nGood-bye World!\n";

end;
```

Section X.4.B. Overloading Logical Operators

Compiler will apply any implicit conversion it can find to operands of logical operators for converting them to boolean before checking for operator overloading.

This is in opposite direction of checking for operator overloading with regard to comparison operators. For logical operators, the logical result is of primary interest and overloading comes secondary to it.

The following example illustrates the order mentioned above.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// Type convertible can reduce itself to boolean.
// Its promotion from boolean is only used for construction.
// ------------------------------------------------------------------ //

struct convertible
      boolean b;

      convertible(void);
      convertible(boolean);
      operator boolean (void);
end;

convertible::convertible(void)
      b = True;
end;

convertible::convertible(boolean bb)
      b = bb;
end;

convertible::operator boolean (void)
      return b;
end;

// ------------------------------------------------------------------ //
// Type convertibleOverload uses operator && to convert
// an instance of int to an instance of convertible.

struct convertibleOverload
      convertible c;

      convertibleOverload(boolean);
      convertible& operator&&(int);
end;

convertibleOverload::convertibleOverload(boolean b) : c(b)
```

```
end;

convertible& convertibleOverload::operator&&(int i)
      return c;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// ------------------------------------------------------------------ //
// (co && coArgument) invokes overloaded && returning a
// reference to an instance of convertible. Note that
// co does not provide conversion to boolean so operator
// overloading is applied.
// The object returned by (co && coArgument) reduces itself
// to a boolean object.
// cvb also reduces itself to an instance of boolean.
// Since both operands are true, the second logical
// operator && evaluates to true.

      int coArgument = 7;
      convertibleOverload co(True);
      convertible cvb;

      if (co && coArgument && cvb)
            output << "(co && coArgument && cvb) is true.\n";
      endif;

      output << "\nGood-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
(co && coArgument && cvb) is true.

Good-bye World!
```

Section X.4.C. Operand Type for Logical expressions

Unlike C, the type of operand for logical operators must be **boolean**, or convertible to **boolean**. The only exceptions are objects of pointer types, which the negation operator accepts as operands for testing their nullity.

Numeric objects and objects of enumeration type are not **boolean**, and unlike C they cannot be used as operands of logical operators. Instead, you need to use the comparison operators. The result of a comparison is an object of **boolean** type and can be used as operand for logical operators.

```
int intObject = 0;
double doubleObject = 0;

// The following will generate error
if (!intObject && !doubleObject) endif;

// This is correct
if (intObject != 0 && doubleObject != 0) endif;
```

Chapter X.5. Increment/Decrement Operators

Increment operator ++ and decrement operator -- can be used as prefix and postfix. For built-in numeric types the location of these operators is significant. As prefix, first the operator is applied to object then the value of the object is used in the expression in which it appears. As postfix, first the object is used in evaluating the expression containing the object, and then the operator is applied to the object.

Compiler does **not** apply any **implicit** conversions to operands of these operators.

Z++ deals with objects rather than literal values of objects. The postfix operators return a temporary object for evaluation in an expression, and then apply the operator to the object itself. However, the prefix forms of these operators, apply the operators to the object, and then return the object, not a temporary. This is illustrated in the following.

```
//IncDec.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      int object = 10;

// object++, returns a temporary of value 10. It then
// increments object to 11. However, operator -- receives
// the temporary of value 10, and decrements it to 9.

      output << --(object++) << '\n'; // prints 9
      output << object << '\n'; // print 11, not 10

// Prefix operators act on object, and return the object
// not a temporary.

      output << --(++object) << '\n'; // prints 11
      output << object << '\n'; // prints 11

      output << "\nGood-bye World!\n";

end;
```

Increment/Decrement operators can be overloaded. **Only for their definition**, the overloading of **postfix** versions use a different symbol, as illustrated below.

Overloaded operators are methods, and as such are invoked on an object without the compiler's intervention. That means, the postfix meaning of these operators with regard to objects of numeric types does not necessarily hold for overloaded versions of these operators. However, in the following example, it seems like the postfix ++ first yields the value of the object, and then applies the incrementation. That is only because the **return** statement is acting on the integer member i++, which returns the current value of the member i **before applying the operator**.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------ //

struct PostOverload
      int i;

      PostOverload(int);
      int operator++(void);
      int operator--(void);
      int operator+@(void); // postfix ++
      int operator-@(void); // postfix --

PostOverload::PostOverload(int n)
      i = n;
end;

int PostOverload::operator++(void)
      return ++i;
end;

int PostOverload::operator--(void)
      return --i;
end;

int PostOverload::operator+@(void)
      return i++;
end;

int PostOverload::operator-@(void)
      return i--;
end;

////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////

entry void main(void)
```

```
        output << "Hello World!\n";

        PostOverload PO(10);

        output << "Result of PO++ is: " << PO++ << '\n'; // 10
        output << "PO.i after PO++ is: " << PO.i << '\n'; // 11

        output << "Result of PO-- is: " << PO-- << '\n'; // 11
        output << "PO.i after PO-- is: " << PO.i << '\n'; // 10

        output << "Result of ++PO is: " << ++PO << '\n'; // 11
        output << "PO.i after ++PO is: " << PO.i << '\n'; // 11

        output << "Result of --PO is: " << --PO << '\n'; // 10
        output << "PO.i after --PO is: " << PO.i << '\n'; // 10

        output << "\nGood-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Result of PO++ is: 10
PO.i after PO++ is: 11
Result of PO-- is: 11
PO.i after PO-- is: 10
Result of ++PO is: 11
PO.i after ++PO is: 11
Result of --PO is: 10
PO.i after --PO is: 10

Good-bye World!
```

Section X.5.A. Inc/Dec of Pointers

The difference between prefix/postfix forms of these operators applies to objects of pointer type. When increment (decrement) operator is applied to a pointer, the size by which the pointer is incremented (decremented) is the size of the object the pointer is pointing to.

The following example shows the usual use of increment operator with regard to pointers.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      char charArray[7];
      charArray = "abcdef"; // initialize array

      char* p = &charArray[0]; // point to first cell

      char Buffer[7];
      char* q = &Buffer[0]; // point to first cell

// First copy each cell, then move pointer forward until 0 is reached

      do *q++ = *p++; enddo(*p == 0);
      *q = 0; // make last cell 0

      string Result = Buffer; // Result == "abcdef"

      output << Result << '\n';

      output << "\nGood-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
abcdef

Good-bye World!
```

Section X.5.B. Inc/Dec of Enumerations

The semantics of prefix/postfix forms of increment/decrement operators apply to objects of enumeration types, as well. The action of increment (decrement) operator on an object of enumeration type is also referred to as the successor (predecessor) function.

The following example demonstrates the use of successor/predecessor functions of enumerations type.

The brackets around an object of enumeration type evaluates to its numeric value.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------- //

enum primes {_three = 3, _five = 5, _seven = 7, _eleven = 11};

////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      primes first = _five;

// second initializes to _five, then first becomes _seven

      primes second = first++;

// first becomes _eleven, initializing third to _eleven

      primes third = ++first;

      output << "first is: " << [first] << '\n'; // prints 11
      output << "second is: " << [second] << '\n'; // prints 5
      output << "third is: " << [third] << '\n'; // prints 11

      output << "\nGood-bye World!\n";

end;
```

Section X.5.C. Inc/Dec of Arrays

Increment/decrement operators are internally overloaded for arrays of numeric types, as well as arrays of pointers and enumeration types. Array operands can be dynamic or static kind. However, the action of postfix operator is identical to the action of the prefix operator in that the operator is applied to the cells of array prior to using the array object in any expression.

Here are some examples.

```
int dim = 3;
short shortDArray[dim](45); // dynamic array
++shortDArray; // cells become 46
shortDArray++; // cells become 47

ulong ulongArray[3](777); // static array

// All cells of both arrays become 778
ulong uArray[3] = ulongArray++;
```

Section X.5.D. Inc/Dec of Bitfields

Increment/decrement operators can also be applied to fields of an object of **bitfields** type. In this case these operators retain their usual semantics as with built-in types. That is, in postfix form first the value of the field is used in evaluating an expression, then the operator is applied to the field.

Note that, regardless of number of bits, fields are unsigned. For instance, decrementing a field of 3 bits wide, with value 0, results in value of the field to become 7.

Chapter X.6. Unary Prefix Operators

Unary prefix operators are the following.

! negation operator was discussed in Logical Operators.
& take address of object.
* de-reference pointer.
~ bit inversion. Invert bits of a **discrete** numeric object.
– minus. Change the sign of a numeric object.

Except for negation (logical) operator, compiler does not apply implicit conversions prior to evaluating the action of a unary operator.

Unary operators **do not modify** their operand. Instead, they yield a **temporary** object for the result of their action.

Unary operators can be overloaded.

Section X.6.A. Address operator &

An object occupies space in its virtual world and therefore it **exists** as an identifiable entity. It follows that an object has a location. The physical memory may use any number of dimensions and coordinate systems for specifying the location of an object. Indeed, an object may reside on another form of storage as the virtual memory extension of the actual physical memory.

Generally a computation refers to objects by their names. When we need to write an algorithm that acts on a set of objects the use of names causes difficulties. Instead, we use an array as a representation of the set of objects under consideration. The **indices** of the array serve the purpose of **coordinates** for locating objects in the set.

Nonetheless, there are times that the ability to use a more general coordinate system to reference objects is desirable and far more elegant. Since each computing device may use its own unique coordinate system, we need an **abstraction** for the literal representation of **coordinate** values of an object. This abstraction is called the **address** of the object. A single literal value, namely 0, is universally taken as an **invalid address**.

Operator & evaluates to the literal representation of coordinates of an object residing in the virtual world of a computer. It is therefore called the **address operator**. The address operator returns a **pointer object** of type pointer to the type of its operand.

Section X.6.B. De-reference operator *

The de-reference operator * is the inverse of the address operator. When applied to a pointer object, the de-reference operator yields the object. Note that operator * **never** returns a **temporary**. It always evaluates to the **actual object** that its operand points to.

```
// Examples
int intObject = 7;
int* intPointer = &intObject;
output << *intPointer; // prints 7, the literal value of intObject

// Operator * is the inverse of operator &
output << *&intObject; // prints 7
```

The following shows how to use C-like pointer to an array.

```
// Make a static array of integers.

int IntSArray[5];
IntSArray[0] = 2;
IntSArray[1] = 3;
IntSArray[2] = 5;
IntSArray[3] = 7;
IntSArray[4] = 11;

// Incorrect assignments resulting in error message from compiler.

//int* ip = IntSArray; // right operand is of type int(5)
//int* ip = &IntSArray; right operand is of type int(5)*

int* ip = (int*) &IntSArray; // This works

// The loop prints: 2, 3, 5, 7, 11.

for (int count = 0; count < 5; count++)
      output << *ip++ << '\n';
endfor;
```

Section X.6.C. Bit Inversion operator ~

Operator ~ returns an object with its bits the opposite of the bits of its operand, **without changing the operand**. The type of object returned is same as the type of operand, which must be a discrete numeric type, from **char** to **long** and their **unsigned** versions.

For inverting the bits of an object use the assignment operator ~= as in x ~= x.

Section X.6.D. Minus operator -

Operator – does not change its operand. Instead it returns an object of same type as its operand, with opposite sign of the literal value of its operand. To change the sign of the object itself use assignment, as in x = -x. There is **no special operator** for this purpose. The operator -= will perform a **subtraction**.

Chapter X.7. Special Operators

Special operators cannot be overloaded. These operators have special meanings, which should not change. Allowing these operators to be redefined by users could change the language, or at least make it hard to understand programs.

Special operators are discussed in sections of this chapter.

Section X.7.A. Scope Operator

Scope operator `::` is used (recursively) in several contexts.

Namespace resolution. The scope operator allows reaching **public** items in nested namespaces. When attached as a prefix to an item it references the global instance of that item.

Structures. Scope operator can be used to reach **public** items of bases from which a structural type is derived.

Collections/Enumeration. The use of scope operator for collections is analogous to its use with structures.

Literal values of objects of type enumeration must start with an underscore. Since nothing else can start with an underscore this makes it easy to identify the type of an enumeration literal. However, at times multiple types may use the same literal value. To resolve the ambiguity the value must be adorned with its type. For nested enumeration types use the scope operator recursively.

Section X.7.B. Derivation Operator

The derivation operator : colon is used for specifying bases from which namespaces, structural types and collections are derived. It is also used for specifying the base in extending an enumeration type.

Section X.7.C. Structure Operators

Besides scope operator and derivation operator mentioned above, structural types use the following operators.

`.` (period) member/method selector.
`->` pointer member/method selector.
`->>` pointer base selector (none-recursive).
`:=` associates a **frame** to a GUI **canvas**.
= specifies type of module.

The pointer operator for selecting bases is similar to pointer operator for selecting members. However, once you reach a base you will need to use scope operator for selecting a member, method or a base of the object you have reached. In other words, operator `->>` is not for recursive use as is the operator `->`. Actually, this is as it should be because bases of a type cannot be pointers to other types, while members can.

Section X.7.D. Comma Operator

In general, semicolon terminates a statement. However, more complex statements use additional keywords to indicate their termination. For instance a **class** definition uses the term **end** followed by a semicolon for its termination. This allows the use of semicolon as a terminator for intermediate statements comprising the definition of a **class**.

The comma operator is mostly used as a separator. For instance, in the definition of a function the parameters are separated with comma. As another example, the literal values for defining an enumeration type are separated with comma. However, in some cases it means more than just a separator and for that reason the term operator is used. Consider the following declaration.

```
int one = 5, two = 10, three = 15;
```

In this example the comma retains the type for all objects. In the conditional statement discussed next comma means slightly more than just a separator.

Section X.7.E. Conditional Expression Operators

Consider the following example.

```
int j = 5, k = 7;
if (j < k) return j; else return k; endif;
```

The **if** statement above is not returning an object. Instead, it is the **return** statements in its legs doing so. The conditional expression is an arithmetic expression that returns an object, as illustrated below.

```
struct test
      int n;

      int operator<(int);
end;

int test::operator<(int m)
      return (n < m) ? n : m;
end;
```

The **?** and **:** separate the legs of a conditional (arithmetic) expression. The arithmetic semantics is significant, as seen from a different version of the previous example.

```
int j = 5, k = 7;
int r = ++(j < k ? j : k); //r will be 6
```

Since the arithmetic expression (statement) as a whole returns an object we can use it as part of an expression, in this case as operand to the increment operator.

Conditional expression cannot return objects of structural type or arrays. However, it can return objects of type pointer, **string**, enumeration and all the numeric types from **char** to **double**. Below is an example for returning strings.

```
string stgOne = "First string.";
string stgTwo = "Second string";

string stgResult = (stgOne <= stgTwo ? "True Leg" : "False Leg");
output << "stgResult is: " << stgResult << '\n'; // prints "True Leg"
```

The comma operator can be used to separate statements in the legs of a conditional expression. However, for each leg the object returned by that leg is the result of the last statement in that leg.

Conditional statements can be **nested** to any level. That is, each leg of a conditional statement can contain other conditional statements.

The **type of objects returned** by the legs of a conditional statement must be the **same**. The expression returns one object for its final result. Thus, one leg cannot return an object of type **integer**, and the other leg return an object of type **double**.

Section X.7.F. Signaling Operators

<- send a signal
? receive a signal

For use of these operators please see Generating and Catching Signals.

Section X.7.G. Size Operator

The operator **size** returns the size of objects of built-in numeric types, strings and dimensions of dynamic arrays. For numeric types it also accepts type name as in `size(long) == 8`. For an object of type **string size** returns the usual length of **string**.

For a dynamic array, say DA, **size**(DA, dim) returns the size of dimension dim. Array dimensions start at 0. That is first dimension is 0, second dimension is 1, etc.

Section X.7.H. Cast Operator

The **cast** specification is illustrated in Explicit Cast. Here, we are discussing the **cast** operator, which takes two operands. Before that, let us mention that Z++ supports C-like casting for **pointers only**. The semantics is identical to that in C. A **pointer cast** instructs the compiler to, temporarily consider a pointer to be of the type being cast to, **without** changing the pointer being cast. Pointer (C-like) cast has its uses.

The **cast** operator returns a temporary object of the type being cast to. For the use of **cast** operator on **pointers** with regard to inheritance tree, see Inheritance and Casting. As the following example illustrates, you can **cast** a structure (**struct**, **class**, **task**, **frame**) object to one of its bases. However, compiler will not let you get a **reference** to the base you are casting to.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//
// Bases for struct baseTwo

struct leftBase
      short s;
end;

struct middleBase
      double b;
end;

struct rightBase
      int i;
end;

//-------------------------------------------------------------------//
// Bases for struct derived

struct baseOne
      double d;
end;

struct baseTwo : leftBase, middleBase, rightBase
      int t;
end;

struct baseThree
      short t;
end;

//-------------------------------------------------------------------//

struct derived : baseOne, baseTwo, baseThree
```

```
	int n;
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
	output << "Hello World!\n";

	derived drv;
	drv::baseTwo::middleBase::b = 5.6;

	middleBase mb = cast(middleBase, drv);

	output << mb.b << '\n'; // prints 5.6

	output << "Good-bye World!\n";
end;
```

The output is as follows.

```
Hello World!
5.6
Good-bye World!
```

One use of **cast** is to remove the **const**. This is illustrated in the following example where the compiler allows assignment to the result of **cast**. Note that the conversion **operator** returns a **const** reference. Usually you would use this in argument passing, rather than making an assignment to it.

```
// Sample.zpp

//--------------------------------------------------------------------//

struct ForReturn
	int i;
end;

struct ReferenceReturn
	ForReturn fr;

	operator const ForReturn& (void);
end;

ReferenceReturn::operator const ForReturn& (void)
	return fr;
end;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
```

```
      ReferenceReturn refRet;
      (cast(ForReturn&, refRet)).i = 17;

end;
```

There are times that the removal of **const** is necessary for avoiding a sizeable code change. However, it also opens up a way of **breaking the laws of privacy**. In the following example direct assignment is made to a **private** member of a **class**. Thus, the removal of **const** must be done with great care.

The **cast** operator evaluates to a **none-constant** reference to the **private** member, allowing direct assignment to it.

```
// Smaple.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

class cheating
      int i;
public:
      cheating(int);
      const int& member(void);
end;

cheating::cheating(int k)
      i = k;
end;

const int& cheating::member(void)
      return i;
end;

////////////////////////////////////////////////////////////////////////
//-------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      cheating ct(int(7)); // member i is initialized to 7
      cast(int&, ct.member()) = 5; // member i changes to 5
      output << ct.member() << '\n'; // prints 5

      output << "Good-bye World!\n";
end;
```

The program prints 5, instead of 7.

```
Hello World!
5
Good-bye World!
```

The last example in this section illustrates casting an array to **char***. The array can be static or dynamic, but it must be a one-dimensional **char** array. The **cast** operator returns a pointer **pointing to the start of array**, as in C-style casting. This is useful when passing arguments to a C++ dynamic library. Note that you can always assign a **char** array to a **string** without casting.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      int dim = 11;
      char dArrayChar[dim];
      dArrayChar[0] = 'I';
      dArrayChar[1] = ' ';
      dArrayChar[2] = 'a';
      dArrayChar[3] = 'm';
      dArrayChar[4] = ' ';
      dArrayChar[5] = 'f';
      dArrayChar[6] = 'i';
      dArrayChar[7] = 'n';
      dArrayChar[8] = 'e';
      dArrayChar[9] = '.';
      dArrayChar[10] = 0;

// Casting array to char*

      char* dap = cast(char*, dArrayChar);
      string stg = dap;
      output << stg << '\n';

// Arrays can be assigned to strings

      string fine = dArrayChar;
      output << fine << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
I am fine.
I am fine.
Good-bye World!
```

Section X.7.I. New Operator

Operator **new** creates a **dynamically** allocated object, and returns a **pointer** to the object it creates. As illustrated in the following example, when the type of operand for **new** is an array, **new** actually returns a pointer to array, **not** to the cell of the array as in C++.

For a static array, the type of pointer returned by **new** takes the sizes of array dimensions into account. Thus, the **typedef** must use the same size for each dimension. This is illustrated by **typedef** IntArray.

For a dynamic array as argument of **new operator**, the **typedef** must be generic. That is, the size of dimension must be left out. This is illustrated by **typedef** dArrayCellPtr.

The **new operator** can be used recursively. The **destroy operator** is very helpful in deleting such sequence of pointers.

The calls to the destructor of cell, in the output of the example, show that the **delete operator** deletes all cells of the array.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

struct cell
      double d;

      cell(double);
      ~cell(void);
end;

cell::cell(double b)
      d = b;
end;

cell::~cell(void)
      output << "cell::~cell(void)\n";
end;

//-------------------------------------------------------------------//

typedef dArrayCellPtr cell[]*;
typedef IntArray int[5];

////////////////////////////////////////////////////////////////////
//-------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";
```

```
        IntArray* ip = new int[5](7);
        output << (*ip)[3] << '\n';

        int dim = 3;
        dArrayCellPtr dacp = new cell[dim](3.7);

        output << (*dacp)[1].d << '\n';

        delete ip;
        delete dacp;

        output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
7
3.7
cell::~cell(void)
cell::~cell(void)
cell::~cell(void)
Good-bye World!
```

Section X.7.J. Delete Operator

Operator **delete** is the opposite of **new operator**. Delete restores the location of memory that **new operator** allocates for an object. The operand of **delete** must be a pointer, and if necessary, **delete** invokes destructor before restoring the space taken by the object pointed to. For an example see **new operator**.

Section X.7.K. Destroy Operator

Operator **destroy** is recursive version of **delete**. The operand for destroy can be a pointer or an array (static or dynamic) of pointers. We will illustrate recursive function of **destroy** via examples. Destroy, descends to end of each sequence of array/pointer combination and deletes the leaves, moving backwards as it deletes pointers.

Unlike **delete**, operator **destroy** does not raise exception on a null pointer. When its operand is a null pointer, **destroy** takes no action, leaving the pointer alone.

In the first example, **destroy** is applied to a dynamic array of two-level pointers, to a static array of two-level pointers, to a structure. As seen from output, all destructors are invoked after applying the **destroy** operator.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------ //

struct value
      int i;

      value(int);
      ~value(void);
end;

value::value(int n) // constructor
      i = n;
end;

value::~value(void)  // destructor
      output << "In value::~value(void)\n";
end;

// ------------------------------------------------------------ //
// typedefs for declaring array object vip

typedef valuePointer value*; // for operand of new operator
typedef valuePointerPointer value**;
typedef ArrayValuePtr valuePointerPointer[3];

// typedef for operand of new operator

typedef ArrayValuePtrPointer ArrayValuePtr*;

////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////

entry void main(void)
```

```
      output << "Hello World!\n";

// sizes for a two dimensional dynamic array

  int first = 3, second = 2;

// vip is a two dimensional dynamic array. Cells of vip are two-level
// pointers to ArrayValuePtr, which is a static array of two-level
// pointers to objects of type value.

  ArrayValuePtr** vip[first][second];

// Initialize vip

  for (int one = 0; one < first; one++)
    for (int two = 0; two < second; two++)

      vip[one][two] = new ArrayValuePtrPointer(new ArrayValuePtr);

      for (int index = 0; index < 3; index++)
        (**vip[one][two])[index] = new valuePointer(new value(777));
      endfor;

    endfor;
  endfor;

// Having used vip, now we can delete all objects, recursively. The
// following will call the destructor of value many times as can be
// seen from the output of destructor.

      output << "\nStart of destroy.\n\n";

  destroy vip;

      output << "\nGood-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!

Start of destroy.

In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
```

```
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)
In value::~value(void)

Good-bye World!
```

In the second example, **destroy** is applied in a variety of situations involving fundamental types. As shown in previous example, in every case objects of fundamental types can be replaced with objects of user-defined types.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

//------------------------------------------------------------------//

typedef IntPointer int*;
typedef IntPointerPointer int**;
typedef ArrayIntPtr IntPointerPointer[3];
typedef ArrayIntPtrPointer ArrayIntPtr*;

////////////////////////////////////////////////////////////////////
//----------------------------------------------------------------//
////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      int first = 3, second = 2;

//------------------------------------------------------------------//

      int*** ipp = new IntPointerPointer(new IntPointer(new int(7)));
      destroy ipp; // deletes 3 pointers

//------------------------------------------------------------------//

      int* aptr[3];
      for (int index = 0; index < 3; index++)
            aptr[index] = new int(47);
      endfor;

      destroy aptr; // deletes 3 pointers

//------------------------------------------------------------------//

      int* daptr[first][second];

      for (int one = 0; one < first; one++)
            for (int two = 0; two < second; two++)
                  daptr[one][two] = new int(97);
```

```
            endfor;
      endfor;

      destroy daptr; // deletes 6 pointers

//-----------------------------------------------------------------------//

      ArrayIntPtr** aip[first][second];

  for (int one = 0; one < first; one++)
    for (int two = 0; two < second; two++)

      aip[one][two] = new ArrayIntPtrPointer(new ArrayIntPtr);
      for (int index = 0; index < 3; index++)
        (**aip[one][two])[index] = new IntPointer(new int(777));
      endfor;

    endfor;
  endfor;

      destroy aip; // deletes 48 pointers

//-----------------------------------------------------------------------//

      output << "Good-bye World!\n";
end;
```

Section X.7.L. Conversion Operator

The conversion **operator** |- converts its right operand, to its left operand. One of the operands must be **string**, and the other must be of numeric type. Thus, if numeric operand is on the right, the conversion **operator** will convert it to an ASCII string. On the other hand, if the numeric operand is on the left, the conversion **operator** will convert the **string** operand on the right to numeric. Below is an example.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      int i = -777;
      string buffer;

      buffer |- i; // convert int to string
      output << buffer << '\n';

      double d = 97.47;

      buffer |- d; // convert double to string
      output << buffer << '\n';

      double result;

      result |- buffer; // convert string to double
      output << result << '\n';

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
-777
97.47
97.47
Good-bye World!
```

Chapter XI. Global Threads

Z++ object-oriented threads are created as instances of task type constructor. There are many situations for which global threads, as opposed to task threads, provide a better abstraction.

In Z++ a **global thread** is simply a global function with **void** return, specified as **thread**, as illustrated below.

```
void globalFun(double)<thread>;
```

The semantics is that, at the point of call a **new thread** is created and the function is invoked in this **new** thread. **The thread that made the call is not blocked**.

The value returned by a global thread is generally unusable. Instead, threads use the signaling mechanism to communicate with one another.

Chapter XI.1. Inter-thread Communication

The first example illustrates the use of signals in inter-thread communication. The main() **entry** point creates a global thread and continues its execution without blocking. The global thread is created and begins its execution.

Later, the main() sends a signal to the **thread**, and waits for the arrival of signal from the thread. The global thread, which was waiting for the arrival of the signal, after receiving the signal, executes its body. At the completion of its execution, before the global thread is **destroyed**, it sends a signal to the main() **entry** point. A global thread is **destroyed** when its execution reaches the **end** of its body.

Here is the first example.

```
// Sample.zpp

#include<iostream.h>
#include<exception.h>

using namespace ioSpace;
using namespace exceptionSpace;

// ------------------------------------------------------------------ //

enum userSignals : signalEventType {
      _SIGNAL_FirstUserSignal,
      _SIGNAL_SecondUserSignal,
      _SIGNAL_ThirdUserSignal
};

// ------------------------------------------------------------------ //
```

```
void globalThread(double d)<thread>

// Wait until signal arrives

      do enddo(signal ? _SIGNAL_FirstUserSignal);

      output << "globalThread. Got: " << d << '\n';

// Generate signal, indicating termination

      signal <- _SIGNAL_SecondUserSignal;

end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Create a thread, without blocking

      globalThread(4.3);

// Send a signal to the thread

      signal <- _SIGNAL_FirstUserSignal;

// Wait until thread sends signal

      do enddo(signal ? _SIGNAL_SecondUserSignal);

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
globalThread. Got: 4.3
Good-bye World!
```

The next example illustrates the simplicity of use of **type** with regard to global threads.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

// ---------------------------------------------------------------- //
```

```
// Extend system signals for thread communication.

enum MyOwnSignals : signalEventType {
      _SIGNAL_TerminateYourself,
      _SIGNAL_TerminationComplete
};

// ------------------------------------------------------------------ //
// Define a thread function type.

type void MyGlobalThreadType(int)<thread>;

// ------------------------------------------------------------------ //
// This is the thread that will be the value
// of instance of MyGlobalThreadType.

void globalThread(int i)<thread>

      output << "From globalThread, i is : " << i << '\n';

      MyOwnSignals mySignal = _SIGNAL_TerminateYourself;

// Wait until parent tells us to terminate

      do enddo(signal ? mySignal);

      output << "globalThread is terminating...\n";

// Tell parent thread we are done

      signal <- _SIGNAL_TerminationComplete;

end;

// ------------------------------------------------------------------ //
// This will receive an instance of MyGlobalThreadType and execute it

int threadReceiver(MyGlobalThreadType t)
      int j = 77;
      output << "Passing : " << j << " to thread.\n";

      t(j); // begin a thread, without blocking

      j = 19;
      output << "Returning : " << j << " from threadReceiver().\n";
      return j;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Create an instance of MyGlblThread
```

```
      MyGlobalThreadType MyGlblThread = globalThread;

      output << "main: Calling threadReceiver() ...\n";

 // Pass MyGlblThread to a function

      int j = threadReceiver(MyGlblThread);

      output << "main: threadReceiver() returned " << j << '\n';

// threadReceiver() has returned but the thread it created
// is still executing. Here we terminte the thread.

      output << "main: Terminating thread.\n";

// Tell thread to terminate

      signal <- _SIGNAL_TerminateYourself;

// Wait until thread terminates

      do enddo(signal ? _SIGNAL_TerminationComplete);

      output << "Good-bye World!\n";
end;
```

Reading the output needs attention to the fact that the example is multi-threaded.

Here is the output.

```
Hello World!
main: Calling threadReceiver() ...
Passing : 77 to thread.
From globalThread, i is : Returning : 1977
 from threadReceiver().
main: threadReceiver() returned 19
main: Terminating thread.
globalThread is terminating...
Good-bye World!
```

Chapter XI.2. Registration of Entire Signals

Process-bounded entire signals require registration. These signals do not cross the boundary of the process that generates them. When a thread of the process generates such a signal, **all threads of the process that have registered** to catch these signals, will **eventually** receive the signal.

One important point illustrated by the example of this chapter is the following. Since the thread (in this example the main) that starts a global thread does not block, it may send an entire signal **too early**. The thread being created needs time to register the entire signals that it is willing to catch. So, if the signal is generated prior to the thread being able to complete its registration, it may **not receive** the signal after the completion of its registration. Z47 will delivers a signal as soon as it is generated to all those threads that have registered for it, **so far**. After that, Z47 **removes** the signal from its queue.

Another point illustrated by the example is that entire signals are eventually delivered. The four global threads each register 3 of 5 entire signals. The main() generates all five signals **only once**, and **waits** to receive a completion signal from each of the four threads. Therefore, the fact that the program **terminates** is because all four global threads receive all the entire signals and **complete** their execution.

```
// EntireSignals.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

// ---------------------------------------------------------------- //

enum MyOwnSignals : signalEventType {
      _SIGNAL_Thread_started_1,
      _SIGNAL_Thread_started_2,
      _SIGNAL_Thread_started_3,
      _SIGNAL_Thread_started_4,
      _SIGNAL_Thread_ended_1,
      _SIGNAL_Thread_ended_2,
      _SIGNAL_Thread_ended_3,
      _SIGNAL_Thread_ended_4
};

enum MyEntireSignals : threadEntireEventType {
      _THREAD_ENTIRE_Signal_1,
      _THREAD_ENTIRE_Signal_2,
      _THREAD_ENTIRE_Signal_3,
      _THREAD_ENTIRE_Signal_4,
      _THREAD_ENTIRE_Signal_5
};

// ---------------------------------------------------------------- //
```

```
void FirstGlobalThread(void)<thread>
        accepts(_THREAD_ENTIRE_Signal_1,
                _THREAD_ENTIRE_Signal_2, _THREAD_ENTIRE_Signal_5)

  signal <- _SIGNAL_Thread_started_1;

  output <<
    "FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_1);

  output <<
    "FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_2.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_2);

  output <<
    "FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_5);

  output << "FirstGlobalThread(): ending the thread.\n";
  signal <- _SIGNAL_Thread_ended_1;

end;

// ------------------------------------------------------------- //

void SecondGlobalThread(void)<thread>
        accepts(_THREAD_ENTIRE_Signal_3,
                _THREAD_ENTIRE_Signal_4, _THREAD_ENTIRE_Signal_5)

  signal <- _SIGNAL_Thread_started_2;

  output <<
    "SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_3.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_3);

  output <<
    "SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_4.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_4);

  output <<
    "SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_5);

  output << "SecondGlobalThread(): ending the thread.\n";
  signal <- _SIGNAL_Thread_ended_2;

end;

// ------------------------------------------------------------- //
```

```
void ThirdGlobalThread(void)<thread>
        accepts(_THREAD_ENTIRE_Signal_1,
                _THREAD_ENTIRE_Signal_2, _THREAD_ENTIRE_Signal_5)

  signal <- _SIGNAL_Thread_started_3;

  output <<
    "ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_5);

  output <<
    "ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_1);

  output <<
    "ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_2.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_2);

  output << "ThirdGlobalThread(): ending the thread.\n";
  signal <- _SIGNAL_Thread_ended_3;

end;

// ------------------------------------------------------------------ //

void FourthGlobalThread(void)<thread>
        accepts(_THREAD_ENTIRE_Signal_3,
                _THREAD_ENTIRE_Signal_4, _THREAD_ENTIRE_Signal_5)

  signal <- _SIGNAL_Thread_started_4;

  output <<
    "FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_3.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_3);

  output <<
    "FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_5);

  output <<
    "FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_4.\n";

  do enddo(signal ? _THREAD_ENTIRE_Signal_4);

  output << "FourthGlobalThread(): ending the thread.\n";
  signal <- _SIGNAL_Thread_ended_4;

end;

//////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
//////////////////////////////////////////////////////////////////////
```

```
entry void main(void)

      output << "Main: Hello World!\n";

// ------------------------------------------------------------------ //
// Create each thread, and wait until the thread responds.

      output << "Main: starting FirstGlobalThread().\n";
      FirstGlobalThread();
      output << "Main: waiting for _SIGNAL_Thread_started_1.\n";
      do enddo(signal ? _SIGNAL_Thread_started_1);

      output << "Main: starting SecondGlobalThread().\n";
      SecondGlobalThread();
      output << "Main: waiting for _SIGNAL_Thread_started_2.\n";
      do enddo(signal ? _SIGNAL_Thread_started_2);

      output << "Main: starting ThirdGlobalThread().\n";
      ThirdGlobalThread();
      output << "Main: waiting for _SIGNAL_Thread_started_3.\n";
      do enddo(signal ? _SIGNAL_Thread_started_3);

      output << "Main: starting FourthGlobalThread().\n";
      FourthGlobalThread();
      output << "Main: waiting for _SIGNAL_Thread_started_4.\n";
      do enddo(signal ? _SIGNAL_Thread_started_4);

// ------------------------------------------------------------------ //
// Now generate all entire signals.

      output << "Main: sending signal _THREAD_ENTIRE_Signal_1.\n";
      signal <- _THREAD_ENTIRE_Signal_1;

      output << "Main: sending signal _THREAD_ENTIRE_Signal_2.\n";
      signal <- _THREAD_ENTIRE_Signal_2;

      output << "Main: sending signal _THREAD_ENTIRE_Signal_3.\n";
      signal <- _THREAD_ENTIRE_Signal_3;

      output << "Main: sending signal _THREAD_ENTIRE_Signal_4.\n";
      signal <- _THREAD_ENTIRE_Signal_4;

      output << "Main: sending signal _THREAD_ENTIRE_Signal_5.\n";
      signal <- _THREAD_ENTIRE_Signal_5;

// ------------------------------------------------------------------ //
// Wait until all threads end.

      output << "Main: waiting for _SIGNAL_Thread_ended_1.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_1);

      output << "Main: waiting for _SIGNAL_Thread_ended_2.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_2);

      output << "Main: waiting for _SIGNAL_Thread_ended_3.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_3);
```

```
      output << "Main: waiting for _SIGNAL_Thread_ended_4.\n";
      do enddo(signal ? _SIGNAL_Thread_ended_4);

// ------------------------------------------------------------------ //

      output << "Main: Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Main: Hello World!
Main: starting FirstGlobalThread().
FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.
Main: waiting for _SIGNAL_Thread_started_1.
Main: starting SecondGlobalThread().
SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_3.
Main: waiting for _SIGNAL_Thread_started_2.
Main: starting ThirdGlobalThread().
ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.
Main: waiting for _SIGNAL_Thread_started_3.
Main: starting FourthGlobalThread().
FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_3.
Main: waiting for _SIGNAL_Thread_started_4.
Main: sending signal _THREAD_ENTIRE_Signal_1.
Main: sending signal _THREAD_ENTIRE_Signal_2.
FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_2.
Main: sending signal _THREAD_ENTIRE_Signal_3.
Main: sending signal _THREAD_ENTIRE_Signal_4.
FirstGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.
SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_4.
FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.
Main: sending signal _THREAD_ENTIRE_Signal_5.
Main: waiting for _SIGNAL_Thread_ended_1.
FirstGlobalThread(): ending the thread.
SecondGlobalThread(): waiting for _THREAD_ENTIRE_Signal_5.
ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_1.
FourthGlobalThread(): waiting for _THREAD_ENTIRE_Signal_4.
Main: waiting for _SIGNAL_Thread_ended_2.
SecondGlobalThread(): ending the thread.
ThirdGlobalThread(): waiting for _THREAD_ENTIRE_Signal_2.
FourthGlobalThread(): ending the thread.
Main: waiting for _SIGNAL_Thread_ended_3.
ThirdGlobalThread(): ending the thread.
Main: waiting for _SIGNAL_Thread_ended_4.
Main: Good-bye World!
```

Chapter XI.3. Registration of Hear Signals

Global threads must register all hear signals they intend to catch. A thread must have a statement for catching every hear signal it registers in order to avoid compile error.

Tell/hear signals are intended for distributed signaling. However, in this example we are using a thread within the same process, rather than two processes on two different nodes. The pattern is the same, though.

There are two hear signals: _SIGNAL_telling_1 and _SIGNAL_telling_2. Main sends the first signal indirectly by calling tellerFun(), but sends the second signal directly.

The thread that registers the signals and catches them is hearFun(). Here too we have slightly complicated the program in order to illustrate the use of **type** for defining a function type. Instead of starting the thread by directly calling hearFun(), we are passing an instance of it to the function beginHearFunThread(), which begins the thread.

Here is the example.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

// ------------------------------------------------------------------ //
// Make some events and hear signals.

enum MyOwnSignals : signalEventType {
      _SIGNAL_Thread_started,
      _SIGNAL_Thread_ended
};

enum MyTellSignals : tellSignalType {
      _SIGNAL_telling_1,
      _SIGNAL_telling_2
};

// ------------------------------------------------------------------ //

struct argumentType
      int n;
      long g;

      argumentType(void);
      argumentType(int, long);
end;

argumentType::argumentType(void)
```

```
      n = 2;
      g = 89;
end;

argumentType::argumentType(int nn, long gg)
      n = nn;
      g = gg;
end;

// ------------------------------------------------------------------ //

int mm = 25;
double dd = 56.12;

void tellerFun(void)

  output << "(tellerFun) Sending values : " << mm << ' ' << dd << '\n';
  tell <- _SIGNAL_telling_1 : mm, dd;

end;

// ------------------------------------------------------------------ //
// All registered hear signals must be heard.

void hearFun(void)<thread>
hears(_SIGNAL_telling_1<int, double>, _SIGNAL_telling_2<argumentType>)

      int m;
      double d;
      argumentType hatp;

// Inform parent we are ready

      signal <- _SIGNAL_Thread_started;

// Wait for a tell signal. We are expecting two
// different signals before termination. However
// we do not know the order in which they may arrive.

      boolean signalOne = False;
      boolean signalTwo = False;

      do
            if (hear ? _SIGNAL_telling_1 : m, d)
                  signalOne = True;
                  output << "(hearFun) Values received from tell : " <<
                            m << ' ' << d << '\n';
            endif;

            if (hear ? _SIGNAL_telling_2 : hatp)
                  signalTwo = True;
                  output << "(hearFun) Received _SIGNAL_telling_2.\n";
                  output << "(hearFun) struct members are : " <<
                            hatp.n << ' ' << hatp.g << '\n';
            endif;

      enddo(signalOne && signalTwo);
```

```
// Tell parent we are terminating.

      signal <- _SIGNAL_Thread_ended;

end;

// ------------------------------------------------------------------ //
// The type defined here, and beginHearFunThread() are
// not necessary. Main can directly call hearFun() and
// start the thread. This is for illustration on use of
// type for defining threads with hears specifications.

type void hearFunType(void)<thread>
hears(_SIGNAL_telling_1<int, double>, _SIGNAL_telling_2<argumentType>);

void beginHearFunThread(hearFunType hft)
      hft(); // start the thread hearFun()
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Instead of next two lines we can just call hearFun()

      hearFunType hfObject = hearFun;
      beginHearFunThread(hfObject);

// Wait until thread completes registration

      do enddo(signal ? _SIGNAL_Thread_started);

// Have tellerFun() send the first signal

      tellerFun();

// Prepare some arguments and send the second
// signal directly.

      argumentType latp(21, 512);
      MyTellSignals tso = _SIGNAL_telling_2;
      tell <- tso : latp;

// Wait until child thread completes

      do enddo(signal ? _SIGNAL_Thread_ended);

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
(tellerFun) Sending values : 25 56.12
(hearFun) Values received from tell : 25 56.12
(hearFun) Received _SIGNAL_telling_2.
(hearFun) struct members are : 21 512
Good-bye World!
```

Chapter XI.4. Disengage Signals

Disengage signals are ordinary signals derived from signalEventType. The terminology is with regard to the context in which these signals are used. A server waits at accept() system call for an incoming connection request. The purpose of disengage signals is the make the server break off the accept() call and respond to the signal. The server can go back to its wait at accept() after responding to the signal, or simply terminate.

The example of this chapter uses Client Server Model and Socket Library from standard library. It is also important to set the IP address correctly, and make sure the port you are using is open, before building this example.

In this example, the signal `_SIGNAL_TerminateServer` is used as disengage signal in the following line.

```
void serviceLoop(void)<thread : _SIGNAL_TerminateServer>
```

Disengage signals, for a global thread, are specified as a comma-separated list of signals following a **colon** after the specification **thread**.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////
// Important. Before compiling please set the ip address for
// creating client to the IP address of machine that is
// running the Server, which could be the same machine.
// Search on "127.0.0.1".
// ---------------------------------------------------------------- //
// Make sure port you are using is open. Here we are using 5001.
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// Include header file for Internet transmission
// (client/server, sockets).
// ---------------------------------------------------------------- //
```

```
#include<transmit.h>
using namespace transmitSpace;

////////////////////////////////////////////////////////////////////////
// Make some signals by extending system signals.
// ------------------------------------------------------------------ //

enum MyServiceSignals : signalEventType {
      _SIGNAL_TerminateServer,
      _SIGNAL_ClientCompleted,
      _SIGNAL_ServerInitializationComplete,
      _SIGNAL_ServerTerminationComplete
};

////////////////////////////////////////////////////////////////////////
// Define a type of global thread that will be used to process
// client requests.
// The name of type is requestProcessorType.
// ------------------------------------------------------------------ //

type void requestProcessorType(uint)<thread>;

////////////////////////////////////////////////////////////////////////
// Here we define the type of object that we will use for
// sending from server to client. We will also overload
// operators << and >> with respect to socket transmission.
// ------------------------------------------------------------------ //

struct transmissionObjectType
      string g;
      char c;
      short s;
      int i;
      double d;

      transmissionObjectType(void);
      transmissionObjectType(int) cast; // explicit cast required

      friend transmissionStreamType&
      operator<<(transmissionStreamType&,
                  const transmissionObjectType&);

      friend transmissionStreamType&
      operator>>(transmissionStreamType&,
                  transmissionObjectType&);

      friend outStream&
      operator<<(outStream&, const transmissionObjectType&);
end;

// Default constructor, for use by Server

transmissionObjectType::transmissionObjectType(void)
      g = "Server has sent :\n";
      c = 'X';
      s = 7;
      i = 47;
```

```
      d = 97.47;
end;

// Constructor for use by client.

transmissionObjectType::transmissionObjectType(int n)
      g = "";
      c = 0;
      s = 0;
      i = n;
      d = 0;
end;

// For sending objects through socket connection

transmissionStreamType&
operator<<(transmissionStreamType& streamInstance,
           const transmissionObjectType& object)

      streamInstance << object.g;
      streamInstance << object.c;
      streamInstance << object.s;
      streamInstance << object.i;
      streamInstance << object.d;

      return streamInstance;
end;

// For receiving objects via socket connection

transmissionStreamType&
operator>>(transmissionStreamType& streamInstance,
           transmissionObjectType& object)

      streamInstance >> object.g;
      streamInstance >> object.c;
      streamInstance >> object.s;
      streamInstance >> object.i;
      streamInstance >> object.d;

      return streamInstance;
end;

// Plain output streaming

outStream&
operator<<(outStream& streamInstance,
           const transmissionObjectType& object)

      streamInstance << object.g << '\n';
      streamInstance << object.c << '\n';
      streamInstance << object.s << '\n';
      streamInstance << object.i << '\n';
      streamInstance << object.d << '\n';

      return streamInstance;
end;
```

```
////////////////////////////////////////////////////////////////////////
// Derive your server class from library class serverType.
// ------------------------------------------------------------------ //
// To your server class, add a new method to serve client requests.
// In this example we have added
//         "void service(requestProcessorType)"
// which takes one argument of type 'requestProcessorType'
// defined earlier.
// ------------------------------------------------------------------ //

class MyServer : serverType

public:

      void service(requestProcessorType); // user-defined
end;

// ------------------------------------------------------------------ //
// When method service() receives an argument for its
// parameter worker, it passes the member transDescriptor
// to it. transDescriptor is mapped to a socket.
// The argument for worker is a thread (defined below). So, the call
// begins a thread to respond to the connection accepted by server.
// ------------------------------------------------------------------ //

void MyServer::service(requestProcessorType worker)
      worker(transDescriptor);
end;

// ------------------------------------------------------------------ //
// This is the global thread function that is
// passed to MyServer::service().
// The thread is cleaned up when the function ends.
// ------------------------------------------------------------------ //

void processRequest(uint descriptor)<thread>

// Make an instance of library class transmissionStreamType for socket
// transmission. descriptor is the connection socket.

      transmissionStreamType tst(descriptor);

// Make an object for sending to client

      transmissionObjectType tot;

// and send it through socket

      tst << tot;

end;

////////////////////////////////////////////////////////////////////////
// This is a user-defined global thread for server's
// accepting connections and invoking a service thread to
// respond to the client's connect call.
```

```
// ------------------------------------------------------------------ //
// The specification of (comma-separated) signals following
// the colon after the term 'thread' is with regard to the
// accept() call. Z47 processor will check for such signals
// even when the thread is blocked on accept() and will
// disengage when the thread  receives such a signal.
// The list of signals is therefore called, the disengage list.
////////////////////////////////////////////////////////////////////

void serviceLoop(void)<thread : _SIGNAL_TerminateServer>

// We have defined the function type requestProcessorType,
// and a global thread processRequest with same signature.
// Here, we make an instance to pass to the method service
// (its worker parameter is the recipient).

      requestProcessorType reqProc = processRequest;

// Make an instance of our server

      MyServer Server;

// Use port number to initialize server

      Server.initialize(5001);

// Tell caller (in this case main) that server
// initialization is complete

      signal <- _SIGNAL_ServerInitializationComplete;

// Now wait for an incoming connect request

      do
            Server.accept();

// If told to terminate, break out of the loop

            if (signal ? _SIGNAL_TerminateServer) break; endif;

// Otherwise make a thread to respond to client,
// and go back to waiting

            Server.service(reqProc);

      enddo;

// Tell caller that server has terminated

      signal <- _SIGNAL_ServerTerminationComplete;
end;

////////////////////////////////////////////////////////////////////
// Let us also make a global thread for the client.
// ------------------------------------------------------------------ //

void client(void)<thread>
```

```
// Make an instance of library class clientType.
// Set IP to Server machine

      clientType Client("127.0.0.1", 5001);

// Establish connection with server

      Client.connect();

// Make an instance of library class transmissionStreamType
// to receive something from server.
// getDescriptor() returns the connection socket.

      transmissionStreamType tst(Client.getDescriptor());

// Create an object for receiving

      transmissionObjectType tot(0);

// Get the object through socket

      tst >> tot;

      output << "Client got :\n\n" << tot;

// Tell caller, main(), we are done

      signal <- _SIGNAL_ClientCompleted;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Begin the server

      serviceLoop();

// Wait until server tells us that it is ready
// for accepting connections

      do
      enddo(signal ? _SIGNAL_ServerInitializationComplete);

// Now begin a client

      client();

// Wait until client is done

      do
      enddo(signal ? _SIGNAL_ClientCompleted);
```

```
// Now tell server to terminate

      signal <- _SIGNAL_TerminateServer;

// Wait until server tells us that it has terminated

      do
      enddo(signal ? _SIGNAL_ServerTerminationComplete);

// We are all done.

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Client got :

Server has sent :

X
7
47
97.47
Good-bye World!
```

Chapter XI.5. The use of Mutex

Type **mutex** is a built-in type with two operations of lock (**operator**++) and unlock (**operator**--). While Z++ facilities reduce the need for using **mutex**, there are times that a mechanism for explicit mutual exclusion specified by user is required.

The following example creates two global threads that increment a single global integer. The threads use a **mutex** object to ensure they take turns in accessing the global integer.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

// ------------------------------------------------------------------ //
// declare a global mutex for use by threads, and
// a global integer for threads to increment.

mutex globalMutex;
int globalInt = 0;
```

```
// ------------------------------------------------------------------ //

enum userSignals : signalEventType {
      _SIGNAL_FirstIncrementSignal,
      _SIGNAL_SecondIncrementSignal,
      _SIGNAL_FirstTerminateSignal,
      _SIGNAL_SecondTerminateSignal,
      _SIGNAL_FirstCompletionSignal,
      _SIGNAL_SecondCompletionSignal
};

// ------------------------------------------------------------------ //
// A thread that increments globalInt.

void firstThread(int n)<thread>

      for (;;)

// Wait until signal arrives

            if (signal ? _SIGNAL_FirstIncrementSignal)

                  globalMutex++; // lock mutex
                  globalInt += n;

                  if (globalInt >= 100)
                        globalMutex--; // unlock mutex
                        signal <- _SIGNAL_FirstCompletionSignal;
                        break; // out of for-loop
                  else globalMutex--; // unlock mutex
                  endif;

            elsif (signal ? _SIGNAL_FirstTerminateSignal)
                  break; // out of for-loop
            endif;

      endfor;

end;

// ------------------------------------------------------------------ //
// A thread that increments globalInt.

void secondThread(int n)<thread>

      for (;;)

// Wait until signal arrives

            if (signal ? _SIGNAL_SecondIncrementSignal)

                  globalMutex++; // lock mutex
                  globalInt += n;

                  if (globalInt >= 100)
                        globalMutex--; // unlock mutex
```

```
                        signal <- _SIGNAL_SecondCompletionSignal;
                        break; // out of for-loop
                  else globalMutex--; // unlock mutex
                  endif;

            elsif (signal ? _SIGNAL_SecondTerminateSignal)
                  break; // out of for-loop
            endif;

      endfor;

end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Starts threads with increments 1 and 2.

      firstThread(1);
      secondThread(2);

      output << "Starting value of globalInt: " << globalInt << '\n';

      do

// Tell threads to increment globalInt

            signal <- _SIGNAL_FirstIncrementSignal;
            signal <- _SIGNAL_SecondIncrementSignal;

// If one thread signals completion, terminate the other thread
// and exit the do-loop.

            if (signal ? _SIGNAL_FirstCompletionSignal)
                  signal <- _SIGNAL_SecondIncrementSignal;
                  break;
            elsif (signal ? _SIGNAL_SecondCompletionSignal)
                  signal <- _SIGNAL_FirstTerminateSignal;
                  break;
            endif;

      enddo;

      output << "Ending value of globalInt: " << globalInt << '\n';

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Starting value of globalInt: 0
Ending value of globalInt: 102
Good-bye World!
```

Chapter XII. Autonomous Agent (travel statement)

Autonomous agent is a process capable of relocating itself. An agent, while executing as a **process** on a node, transports itself to a different node and continues its execution, as it would have on the node where it came from.

The value of an autonomous agent for computational purposes lies in the fact that an agent is a **process**. Sending an instance of a **class** to another node is not of any more value than sending instances of fundamental types.

A Z47 process is a Z++ program in execution, and as such it is a very complex object. A design that provides a pattern involving several keywords and housekeeping is not practical. Instead, Z++ linguistic abstraction for transporting an agent is a simple statement. The entire work is done by, Z++ compiler and the Z47 Processor.

The Z++ **travel** statement transports an autonomous agent, as shown below.

travel ip-address;

The ip-address is a string like "192.164.1.4", the address of destination node. After the execution of the **travel** statement on current node, the agent will continue its execution on the destination node with statement following the **travel** statement. The execution of agent at current node is terminated. The simple view of transportation is that, **the process goes on the waiting queue at current node but wakes up on another node**.

An agent maintains its state as it moves from node to node. Since it is capable of communicating with other nodes and databases, it is possible to design many unforeseen applications around the notion of an autonomous agent. For instance, an agent can periodically check on certain data in the nodes of a system. A system (such as a factory) may utilize many computers, each of which maintaining certain data. Instead of direct hardwire connection to physical viewing devices, an agent can verify the data and report any issues. An airport may be able to use agents to automatically guide arriving airplanes. The agent sent to the airplane will be more aware of resources and specifics of the airport, at any moment, in landing the plane and relinquishing control to the pilot.

Chapter XII.1. Travel statement and entry points

A program sprinkled with **travel** statements is not understandable. The set of **entry** points of a program are few and easily recognizable. Furthermore, the state of the program is more understandable at a point in an **entry** point than in a nested sequence of calls. It is true that engineer is not responsible for transporting the agent. Nonetheless, the **engineer must be able to understand what is being transported**.

In a Z++ program, the **travel** statement can only appear in an **entry** point.

Even in this restricted form, one can create a program too difficult to understand at the point where the **travel** statement appears. The **travel** statement can appear in a deeply nested sequence of scopes of selections, iterations and (exception) layers with resumption. In each of these scopes many objects may be created and the states of other objects could change. It is important that the engineer be able to visualize the state of the agent at the point of being transported, that is, the **travel** statement.

Chapter XII.2. One travel statement per entry point

As another aid to understanding the state of an agent at the time of transportation, the Z++ compiler allows **one single travel** statement per **entry** point. This restriction does not limit the number of locations for agent to move to. The ip-address is a **string**, which can be set in accordance to program conditions, and then passed as argument to the **travel** statement.

Understanding of the state of agent once it is relocated is of fundamental importance. Otherwise the ability to throw an agent to another location is of little value, if any.

Chapter XII.3. States of global objects

The states of objects local to the **entry** point in which the **travel** statement executes are recovered exactly as they were just before the execution of the **travel** statement. However, the states of **global** objects are recovered at their **initialization** stage, at the start of program execution. Therefore, in presence of **travel** statement an **entry** point should not rely on the states of global objects.

Normally, global objects are created for such things as input/output. Their states are not relevant to the execution of the program, as in making decisions with regard to selection statements. At any rate, an **entry** point with **travel** statement should only **rely on states of its local objects**.

Chapter XII.4. Types of objects transported

Recovering states of objects of certain types is not necessary for the utility of autonomous agents for solving problems. In fact, a type like task reduces the ability of an engineer to understand the state of an agent at the point of transport. An agent can start a task and complete its execution before traveling. In this case, the task object would be local to a function called from the **entry** point that makes the **travel**.

Specifically, the types of objects whose states are recovered are all numeric types, from **char** to **double**, **string** and **enumerations**. In addition, the states of (static) **arrays** and **classes** of such objects are recovered (**recursively**). The term recursive is used to indicate that **classes can be derived, have members of type of arrays of classes**, etc.

Specifically pointers and dynamic arrays are not supported. The compiler will report the unsupported cases.

Types that do not admit public copy constructor such as **task**, **frame** and module (defining a component via **class/task**). Compiler will not allow objects of this type in the **entry** point with **travel** statement.

Chapter XII.5. Multi-threaded and Component Agents

An agent can be multi-threaded. However, at the point of execution of the **travel** statement all threads (except the one executing the **travel** statement) must have ended. Otherwise, Z47 will raise exception _EXCEPTION_Travel_TooManyThreads and the agent will not be transported.

The **entry** point containing the **travel** statement could call functions that initiate global threads or **task** objects. The engineer must ensure that all such threads **end** before the **travel** statement is executed.

A Z++ program can load an agent (a Z++ program with **travel** statement). This causes no problem. At some point the agent will make its travel. The program that had loaded the agent should not make further calls to its **entry** points after it has been transported.

However, there is a different scenario that needs care. An agent can also load other components, but at the point of execution of **travel** statement, all components must have been unloaded (terminated). Otherwise, Z47 will not transport the agent, and will raise the exception _EXCEPTION_Travel_LiveLoadedComponent

The compiler is of no help for these types of errors. An engineer dealing with agents must ensure that all threads initiated by the **agent**, and all components loaded by the **agent**, have terminated at the point of execution of the **travel** statement, by that **agent**.

Chapter XII.6. Exceptions raised by travel statement

The **travel** statement may raise the following exceptions. When the **travel** statement raises an exception, the transport of agent is **not done**. You may be able to repair the cause of exception and **try again**.

`_EXCEPTION_ConnectError` is raised when connection to the receiving end (server) fails.

`_EXCEPTION_MemoryException` is raised when the receiving end reports insufficient memory to accept the agent.

Communication problems for sending and receiving data are signaled with following exceptions.

```
_EXCEPTION_SendError
_EXCEPTION_ReceiveError
```

We also mentioned the following exceptions in previous section:

_EXCEPTION_Travel_TooManyThreads
_EXCEPTION_Travel_LiveLoadedComponent

Note that, when agent is loaded as component by another program, and you wish to catch exceptions in that program instead of in agent the rules remain the same. Agent as a component reports all its exception to its parent as:

_EXCEPTION_ComponentRaisedException

It is better to catch agent's exceptions in agent itself, and leave the above exception for unexpected abnormal situations that may arise, so the parent will take care of such cases.

Chapter XII.7. Autonomous Agent Examples

The examples of this section show transport of objects of complex datatypes and the nesting of scopes of iteration, selection and exception (**layer** scope). The examples show the simplicity of linguistic abstraction of **travel** statement versus the complexity of a program being transported. However, the intent of using autonomous agents is to solve specific problems, which is not the purpose of these examples.

Section XII.7.A. Scopes and Datatypes

The travel statement in this example is in a case leg of a **switch** statement (selection) nested in a do-loop (iteration).

The local datatypes are quite complex. The type `localObjectType` is derived from two other types (multiple-inheritance). The bases are further derived from other types, which also have members of **class** type. In addition, an object of array of `localObjectType` is also transported.

Object of fundamental types, including **string** and **enum**, as well as arrays of such types are also transported.

```
// Example.zpp

#include<stream.h>
using namespace fileSpace;

////////////////////////////////////////////////////////////////////////
//
// For this example, please note the following.
//
// The Z++ Internet Server must be up and running.
//
// The IP = "192.168.x.x" is only a placeholder.
// Before building this program, change the IP address to the
// machine's IP that is running the Z++ Internet Server.
//
// The string outDir must be set to a directory where you
// have permissions to create new files.
//
////////////////////////////////////////////////////////////////////////

enum TravelStates {_travel_Before, _travel_After, _travel_End};

// ------------------------------------------------------------------ //

string outDir = "An existing directory";

////////////////////////////////////////////////////////////////////////

struct baseBaseType
      double d;
      string n;
end;

// ------------------------------------------------------------------ //

struct memberBaseType
      string n;
end;
```

```
// ------------------------------------------------------------------ //

struct leftBaseType
      memberBaseType m;
      int i;
end;

// ------------------------------------------------------------------ //

struct rightBaseType : baseBaseType
      string n;
      short s;
end;

// ------------------------------------------------------------------ //
// This is the type used in the example. The above types are for
// its bases and members of bases.
// ------------------------------------------------------------------ //

struct localObjectType : leftBaseType, rightBaseType
      int i;
      double d;
      string n;

      localObjectType(int, double, string);
end;

localObjectType::localObjectType(int t, double b, string g)
      i = t;
      d = b;
      n = g;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      string info;
      string outname;
      string address = "192.168.x.x"; // use correct IP address here

      TravelStates stateOfTravel = _travel_Before;
      boolean Done = False;

// ------------------------------------------------------------------ //
// The states of these local objects will be recovered at destination.
// ------------------------------------------------------------------ //

      int IntArray[3](777);
      TravelStates StateArray[3](_travel_End);
      string StgArray[4]("StgArrayCell");

      localObjectType lot(71, 88.99, "default value");
      localObjectType alot[3](71, 88.99, "default value");
```

```
// ------------------------------------------------------------------ //
// In this example, travel statement appears in a case leg of
// switch statement which is nested in a do-loop.
// ------------------------------------------------------------------ //

      do
            switch(stateOfTravel)

// ------------------------------------------------------------------ //

                  case _travel_Before:
                        info = "We have not traveled, yet.\n";

                        outname = outDir + "TravelBefore.txt";

// Setting and changing value of object lot.

                        lot.i = 7;
                        lot.n = "lot.n";
                        lot.d = 23.21;
                        lot::leftBaseType::m.n =
                                    "lot::leftBaseType:m.n";

                        lot::leftBaseType::i = 55;

                        lot::rightBaseType::n =
                                    "lot::rightBaseType::n";

                        lot::rightBaseType::s = 11;
                        lot::rightBaseType::baseBaseType::d = 11.33;

                        lot::rightBaseType::baseBaseType::n =
                              "lot::rightBaseType::baseBaseType::n";

// Except for some values of the first cell of array object alot,
// we will change the values of all cells.

                        //alot[0].i = 47;
                        //alot[0].d = 97.47;
                        //alot[0].n = "alot[0].n";

                        alot[0]::leftBaseType::m.n =
                              "alot[0]::leftBaseType::m.n";

                        alot[0]::leftBaseType::i = 32;

                        alot[0]::rightBaseType::n =
                              "alot[0]::rightBaseType.n";

                        alot[0]::rightBaseType::s = 13;

                        alot[0]::rightBaseType::baseBaseType::d =
                                    47.97;

                        alot[0]::rightBaseType::baseBaseType::n =
                           "alot[0]::rightBaseType::baseBaseType.n";
```

```
                        alot[1].i = 4;
                        alot[1].d = 4.7;
                        alot[1].n = "alot[1].n";

                        alot[1]::leftBaseType::m.n =
                               "alot[1]::leftBaseType::m.n";

                        alot[1]::leftBaseType::i = 2;

                        alot[1]::rightBaseType::n =
                                    "alot[1]::rightBaseType.n";

                        alot[1]::rightBaseType::s = 3;
                        alot[1]::rightBaseType::baseBaseType::d = 9.7;

                        alot[1]::rightBaseType::baseBaseType::n =
                              "alot[1]::rightBaseType::baseBaseType.n";

                        alot[2].i = 6;
                        alot[2].d = 6.3;
                        alot[2].n = "alot[2].n";

                        alot[2]::leftBaseType::m.n =
                                  "alot[2]::leftBaseType::m.n";

                        alot[2]::leftBaseType::i = 76;

                        alot[2]::rightBaseType::n =
                                   "alot[2]::rightBaseType.n";

                        alot[2]::rightBaseType::s = 88;
                        alot[2]::rightBaseType::baseBaseType::d = 3.8;

                        alot[2]::rightBaseType::baseBaseType::n =
                              "alot[2]::rightBaseType::baseBaseType.n";

// Let us change the first two cells of StateArray.

                        StateArray[0] = _travel_Before;
                        StateArray[1] = _travel_After;

// ------------------------------------------------------------------ //

                  case _travel_After:
                        info =
                          "We have traveled to our destination.\n";

                        outname = outDir + "TravelAfter.txt";

// Let us take off to destination

                        travel address;

// The else leg executes after traveling to destination.

                  else Done = True;
```

```
            endswitch;

// ------------------------------------------------------------------ //
// The boolean Done becomes true after agent transport and
// writing to file "TravelAfter.txt".
// ------------------------------------------------------------------ //

            if (!Done)

// Increment the switch argument to its successor

                  stateOfTravel++;

                  FileStreamType outfile;
                  outfile.setOutput();
                  outfile.open(outname);
                  outfile << info; // Write header info to file

// Write values of members/bases of lot.

                  outfile << "Values of lot:\n" << '\n';

                  outfile << lot.i << '\n';
                  outfile << lot.n << '\n';
                  outfile << lot.d << '\n';
                  outfile << lot::leftBaseType::m.n << '\n';
                  outfile << lot::leftBaseType::i << '\n';
                  outfile << lot::rightBaseType::n << '\n';
                  outfile << lot::rightBaseType::s << '\n';
            outfile << lot::rightBaseType::baseBaseType::d << '\n';
            outfile << lot::rightBaseType::baseBaseType::n << '\n';

// Write values of cells of array alot

                  outfile << "\nValues of array alot[]:\n" << '\n';

                  outfile << alot[0].i << '\n';
                  outfile << alot[0].d << '\n';
                  outfile << alot[0].n << '\n';
                  outfile << alot[0]::leftBaseType::m.n << '\n';
                  outfile << alot[0]::leftBaseType::i << '\n';
                  outfile << alot[0]::rightBaseType::n << '\n';
                  outfile << alot[0]::rightBaseType::s << '\n';
            outfile << alot[0]::rightBaseType::baseBaseType::d << '\n';
            outfile << alot[0]::rightBaseType::baseBaseType::n << '\n';

                  outfile << alot[1].i << '\n';
                  outfile << alot[1].d << '\n';
                  outfile << alot[1].n << '\n';
                  outfile << alot[1]::leftBaseType::m.n << '\n';
                  outfile << alot[1]::leftBaseType::i << '\n';
                  outfile << alot[1]::rightBaseType::n << '\n';
                  outfile << alot[1]::rightBaseType::s << '\n';
            outfile << alot[1]::rightBaseType::baseBaseType::d << '\n';
            outfile << alot[1]::rightBaseType::baseBaseType::n << '\n';
```

```
                    outfile << alot[2].i << '\n';
                    outfile << alot[2].d << '\n';
                    outfile << alot[2].n << '\n';
                    outfile << alot[2]::leftBaseType::m.n << '\n';
                    outfile << alot[2]::leftBaseType::i << '\n';
                    outfile << alot[2]::rightBaseType::n << '\n';
                    outfile << alot[2]::rightBaseType::s << '\n';
               outfile << alot[2]::rightBaseType::baseBaseType::d << '\n';
               outfile << alot[2]::rightBaseType::baseBaseType::n << '\n';

// Values of array of integers, IntArray.

                    outfile << "\nValues of IntArray:\n" << '\n';

                    outfile << IntArray[0] << '\n';
                    outfile << IntArray[1] << '\n';
                    outfile << IntArray[2] << '\n';

// Values of array of enumerations, StateArray.

                    outfile << "\nValues of StateArray:\n" << '\n';

                    outfile << [StateArray[0]] << '\n';
                    outfile << [StateArray[1]] << '\n';
                    outfile << [StateArray[2]] << '\n';

// Values of array of strings, StgArray.

                    outfile << "\nValues of StgArray:\n" << '\n';

                    outfile << StgArray[0] << '\n';
                    outfile << StgArray[1] << '\n';
                    outfile << StgArray[2] << '\n';
                    outfile << StgArray[3] << '\n';

                    outfile.close();

               endif;

          enddo(Done);

end;
```

The output of this program is two identical files, one at source and the other at destination. Below is the **output at destination, the file "TravelAfter.txt"**.

```
We have traveled to our destination.
Values of lot:

7
lot.n
23.21
lot::leftBaseType:m.n
55
lot::rightBaseType::n
11
```

```
11.33
lot::rightBaseType::baseBaseType::n

Values of array alot[]:

71
88.99
default value
alot[0]::leftBaseType::m.n
32
alot[0]::rightBaseType.n
13
47.97
alot[0]::rightBaseType::baseBaseType.n
4
4.7
alot[1].n
alot[1]::leftBaseType::m.n
2
alot[1]::rightBaseType.n
3
9.7
alot[1]::rightBaseType::baseBaseType.n
6
6.3
alot[2].n
alot[2]::leftBaseType::m.n
76
alot[2]::rightBaseType.n
88
3.8
alot[2]::rightBaseType::baseBaseType.n

Values of IntArray:

777
777
777

Values of StateArray:

0
1
2

Values of StgArray:

StgArrayCell
StgArrayCell
StgArrayCell
StgArrayCell
```

Section XII.7.B. Nested Exception Scopes

This examples illustrates the transport of nested **layer** (exception) scopes.

```
// Example.zpp

#include<stream.h>
using namespace fileSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
//
// For this example, please note the following.
//
// The Z++ Internet Server must be up and running.
//
// The IP = "192.168.x.x" is only a placeholder.
// Before building this program, change the IP address to the
// machine's IP that is running the Z++ Internet Server.
//
// The string outname must be full-pathname of file to create.
// You should have permissions to create new files in the directory.
//
////////////////////////////////////////////////////////////////////////

// ------------------------------------------------------------------ //
// Extend the set of system exceptions to include
// user-defined exceptions

enum userExceptionEvents : exceptionEventType {
    _FirstUserException,
    _SecondUserException
    };

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void) throws(_SecondUserException)

      string outname = "Full-pathname-for-file";
      int anInt = 27;
      double aDouble = 47.97;

      userExceptionEvents userExceptionValue;
      exceptionEventType systemExceptionValue;

// ------------------------------------------------------------------ //
// In this example, the travel statement appears in a nested
// scope, including two layers of exception. After travel is done,
// the complete state of the agent is recovered, including exceptions.
// ------------------------------------------------------------------ //

      layer<exceptionEventType>          //outer exception layer

            layer<userExceptionEvents>     //inner exception layer
```

```
// ------------------------------------------------------------------ //
// If exception does occur in this layer, userExceptionValue
// will be changed to _SecondUserException in its handler.
// It is not expected to happen, though.
// ------------------------------------------------------------------ //

                userExceptionValue = _FirstUserException;

// ------------------------------------------------------------------ //
// Let us just make another scope, for no reason, and then
// travel from inside it.
// Set the IP address to your machine's IP.
// ------------------------------------------------------------------ //

                {
                      short st = 7;
                      travel "192.168.x.x"; // Let us take a trip
                      anInt = st; // we are now at destination!
                }

                if (anInt != 7) raise _SecondUserException; endif;

            handler                         //inner layer

                  case _FirstUserException .. _SecondUserException:
                        userExceptionValue = _SecondUserException;

            endlayer;                       //inner layer

            int i = 17;
            i /= 0;    //cause _EXCEPTION_DivisionByZero to happen

      handler                           //outer layer

            case _EXCEPTION_DivisionByZero:
                  systemExceptionValue = _EXCEPTION_DivisionByZero;

      endlayer;                         //outer layer

// ------------------------------------------------------------------ //
// Let us record a few things to see what actually happened.
// ------------------------------------------------------------------ //

      FileStreamType outfile;
      outfile.setOutput();
      outfile.open(outname);

      outfile << "Traveled to destination. Now ending execution.\n";
      outfile << anInt << '\n';
      outfile << aDouble << '\n';

   if (systemExceptionValue == _EXCEPTION_DivisionByZero)
      outfile << "Exception _EXCEPTION_DivisionByZero was caught.\n";
   endif;

   if (userExceptionValue == _FirstUserException)
      outfile << "Exception _SecondUserException was not generated.\n";
```

```
   endif;

      outfile.close();

end;
```

The output of this example is as follows.

```
Traveled to destination. Now ending execution.
7
47.97
Exception _EXCEPTION_DivisionByZero was caught.
Exception _SecondUserException was not generated.
```

Section XII.7.C. Handling Agent Exceptions

In this example, we cause an exception to occur during transport of agent. The cause of exception is repaired, and (using resumption) we repeat the transport.

This example creates two files. TravelLogfile.txt is created at source (where the agent is initially running), and TravelOutput.txt is created at destination, after transport.

```
// Example.zpp

#include<stream.h>
using namespace fileSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////
//
// For this example, please note the following.
//
// The Z++ Internet Server must be up and running.
//
// The IP = "192.168.x.x" is only a placeholder.
// Before building this program, change the IP address to the
// machine's IP that is running the Z++ Internet Server.
//
// The string logfile is filename in a directory where
// you have permissions to create new files on the machine
// prior to transport, and outfile is filename
// after the transport with similar permissions as for logfile.
// outfile and logfile are full-pathname of files.
//
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// The travel statement may raise the following exceptions.
//
//     _EXCEPTION_ConnectError
//     _EXCEPTION_MemoryException
//     _EXCEPTION_SendError
//     _EXCEPTION_ReceiveError
//
// If travel does raise an exception, the agent will not
// actually travel to its destination. Nothing will happen
// at the destination end, but the agent will crash unless
// you do something about the exceptions.
//
////////////////////////////////////////////////////////////////////

// ---------------------------------------------------------------- //
// You may have a list of permissible destinations and
// alternatives with more sophisticated logic to choose a
// destination.
// Here, we make firstDestination invalid to
```

```
// cause _EXCEPTION_ConnectError. secondDestination is a valid
// address to reach the Z++ Internet Server.
// ------------------------------------------------------------- //

string firstDestination = "12.166.5.0"; // use invalid address here
string secondDestination = "192.168.1.64"; // use correct address here

// ------------------------------------------------------------- //
// Library complex objects, like stream objects should be global.
// ------------------------------------------------------------- //

FileStreamType logfile;
FileStreamType outfile;

////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////

// Remark. It will take a while until the first connect fails,
// which may make it appear as if the program is stuck.
// Wait until it responds. The response time is not the same
// on all platforms.

entry void main(void)

      string outname = "~/TravelOutput.txt";
      string logname = "~/TravelLogfile.txt";

// We try an invalid address first

      string destination = firstDestination;

// This is for logging the exception at local site

      logfile.setOutput();

// ------------------------------------------------------------- //

      layer<exceptionEventType>

// causes _EXCEPTION_ConnectError on first attempt

            travel destination;

      handler

            case _EXCEPTION_ConnectError:
                  logfile.open(logname);
      logfile << "Caught _EXCEPTION_ConnectError while traveling to ";
                  logfile << destination << '\n';
                  logfile << "Trying another destination\n";
                  logfile.close();

// repair the problem

                  destination = secondDestination;
                  repeat; // and try again
```

```
            case _EXCEPTION_MemoryException:
                  logfile.open(logname);
logfile << "Caught _EXCEPTION_MemoryException while traveling to ";
                  logfile << destination << '\n';
                  logfile << "Trying another destination\n";
                  logfile.close();
                  destination = secondDestination;
                  repeat;

            case _EXCEPTION_SendError:
                  logfile.open(logname);
      logfile << "Caught _EXCEPTION_SendError while traveling to ";
                  logfile << destination << '\n';
                  logfile << "Trying another destination\n";
                  logfile.close();
                  destination = secondDestination;
                  repeat;

            case _EXCEPTION_ReceiveError:
                  logfile.open(logname);
      logfile << "Caught _EXCEPTION_ReceiveError while traveling to ";
                  logfile << destination << '\n';
                  logfile << "Trying another destination\n";
                  logfile.close();
                  destination = secondDestination;
                  repeat;

      endlayer;

// ------------------------------------------------------------------ //
// Let us record that we actually traveled. This is
// happening at destination.
// ------------------------------------------------------------------ //

      outfile.setOutput();
      outfile.open(outname);

      outfile << "Travel to " << destination << " was successful.\n";

      outfile.close();

end;
```

The output to TravelLogfile.txt is as follows.

```
Caught _EXCEPTION_ConnectError while traveling to 12.166.5.0
Trying another destination
```

The output to TravelOutput.txt is as follows.

```
Travel to 192.168.1.64 was successful.
```

Section XII.7.D. Handling Agent Exceptions - 2

This is another example of resumption, where agent is loaded as component by another program. The program calls an **entry** point of agent, which has **travel** statement. Agent starts a global **thread** that it must terminate before traveling.

This is a simple program that loads agent as component.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

//-------------------------------------------------------------------//

struct AgentModule = "path to /SampleAgent.zxe"
      external void FirstEntry(void);
end;

////////////////////////////////////////////////////////////////////////
//-------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Sample: Hello World!\n";
      output << "Sample: Loading SampleAgent....\n";
      AgentModule AM;
      output << "Sample: Calling SampleAgent::FirstEntry()....\n";
      AM.FirstEntry();
      output << "Sample: Good-bye World!\n";
end;
```

This is the source for Agent.

```
// SampleAgent.zpp

#include<stream.h>
using namespace fileSpace;

#include<exception.h>
using namespace exceptionSpace;

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

enum userSignals : signalEventType {
      _SIGNAL_TerminateThreadSignal,
      _SIGNAL_ThreadTerminatedSignal
};

////////////////////////////////////////////////////////////////////////
```

```
void fun(void)<thread>

// Wait until signal arrives

      do enddo(signal ? _SIGNAL_TerminateThreadSignal);

// Generate signal, indicating termination

      signal <- _SIGNAL_ThreadTerminatedSignal;

end;

////////////////////////////////////////////////////////////////////////

entry void main(void)
// not used.
end;

////////////////////////////////////////////////////////////////////////
// This is called by owner of agent.
////////////////////////////////////////////////////////////////////////

entry void FirstEntry(void)

      string address = "192.168.0.3"; // use correct IP address here
      string outname = "Path to file /TravelOutput.txt";

      output << "Agent: Starting global thread...\n";

      fun(); // This starts a global thread

      layer<exceptionEventType>

            output << "Agent: Starting Travel...\n";

            travel address; // Causes exception

      handler

            case _EXCEPTION_Travel_TooManyThreads:
output << "Caught exception: _EXCEPTION_Travel_TooManyThreads...\n";

// Now terminate the thread, wait until it tells us it has terminated,
// and try again.

                  signal <- _SIGNAL_TerminateThreadSignal;
                  do enddo(signal ? _SIGNAL_ThreadTerminatedSignal);
                  output << "Agent: Repeating Travel...\n";
                  repeat;

      endlayer;

      FileStreamType outfile;
      outfile.setOutput();
      outfile.open(outname);
      outfile << "Traveled to destination, successfully.\n";
```

```
      outfile.close();

end;
```

The output of Sample.zpp goes to consol, as following.

```
Sample: Hello World!
Sample: Loading SampleAgent....
Sample: Calling SampleAgent::FirstEntry()....
Agent: Starting global thread...
Agent: Starting Travel...
Caught exception: _EXCEPTION_Travel_TooManyThreads...
Agent: Repeating Travel...
Sample: Good-bye World!
```

The output of agent is written to file TravelOutput.txt at destination:

```
Traveled to destination, successfully.
```

Appendices

Among other topics, appendices cover all library classes and functions and the preprocessor commands. Complete examples and for simpler cases code fragments illustrate each topic.

Appendix I. Structure of a Z++ program

The **smallest** Z++ program is one single line.

```
// tiny.zpp

entry void main(void) end;
```

Next example is a small program with output.

```
// small.zpp

#include<iostream.h>
using namespace ioSpace;

entry void main(void)
      output << "Hello World!\n";
      output << "Good-bye World!\n";
end;
```

The output of this program is as follows.

```
Hello World!
Good-bye World!
```

A Z++ program must have at least **one entry** point named main, with the **signature** shown here. However, a Z++ program can have any number of **entry** points, with any signature, **scattered in several files of a project**.

Type definitions and declaration of global objects can appear anywhere in between **entry** points. However, for readability reasons, it may be better to collect all **entry** points in a single file.

An **entry** point cannot be **prototyped**, nor it can be **called** from **within** the program.

When a program is executed directly, after initialization the Loader makes a jump to the main **entry** point. On the other hand, when a program is loaded as a component by another program, the Loader **only performs the initialization**. Instead, the loading program invokes the **entry** points of the component.

When a Z++ program is intended for use as a component, the main **entry** point can be used to print a message in case the program is executed directly. Generally the message should inform the user that the program is intended for use as a component.

Appendix II. Preprocessor Commands

Preprocessing is the phase before parsing. The preprocessor combines all files of a project into one single file and passes it to the parser. In preprocessing phase, the preprocessor commands direct its actions. The preprocessor removes its commands as it executes them so that the parser does not see the preprocessor commands.

All preprocessor commands must be preceded with the # symbol, and without the terminating semi-colon.

A preprocessor **identifier** is a Z++ identifier, which can be **preceded** with one or more **underscore** characters.

The **macro** command is illustrated in the next appendix. Here we discuss the commands **include**, **define**, **undef**, **if**, **ifnot**, **elsif**, **elsifnot**, **else** and **endif**.

There is **no negation operator** during preprocessing. Instead, you will need to use **ifnot** and **elsifnot**.

Appendix II.1. The include command

This command instructs the preprocessor to load and insert a file into its output file to the parser. The file being included is preprocessed as it is inserted into the output file.

The **include** command only accepts files **with extension h**, as "filename.h". A file with extension h is called a **header** file.

The **location** of a header file to include can be indicated as follows.

```
#include<filename>
```

When filename is between < and >, the header file will be loaded from the **system directory**, which was set during installation.

```
#include"filename"
```

When filename is between double-quotes, a full path is required. However, **Z++ Visual** allows users to specify paths for inclusion, in which case the preprocessor will look for filename in the specified directories.

You can indicate **subdirectories** of specified directories for the **include** command. For instance, when filename is in a subdirectory of a directory that you have specified via **Z++ Visual**, you can write the following.

```
#include"subdirectoy-name/filename"
```

Appendix II.2. The define command

The argument for the **define** command is a **preprocessor** identifier. All identifiers for the conditional commands are **false** by default. The **define** command sets the value of its argument to **true**.

```
#define Some_label
```

This will set the value of `Some_label` to **true**.

Appendix II.3. The undef command

The **undef** command sets the value of its argument to false, regardless of its value before passing it to the **undef** command.

```
#undef _Another_label
```

This will set the value of `_Another_label` to **false**.

Appendix II.4. The if-else-endif structure

This structure starts with the **if** command, and ends with **endif** command. Optionally, in between it can have several **elsif**, **elsifnot** but a single **else** just before **endif**.

The arguments to **if**, **elsif** and **elsifnot** are preprocessor identifiers. All identifiers are **false** by default. The truth or falsity of identifiers can be set by the commands **define** and **undef**.

Appendix II.5. The ifnot-else-endif structure

This structure starts with **ifnot** command, and is otherwise identical to the **if** structure. A pattern to avoid including a **header** file more than once is as follows.

```
#ifnotdef Some_Id
#define Some_Id

// The statements in the file

#endif
```

When the **include** command inserts this file for the first time `Some_Id` will be false unless you set it to true via the **define** command. Assuming it is false, the statements following the command **ifnotdef** will be included until the command **endif**. However, this will execute the command **define** and set `Some_Id` to **true**. So, the next **include** command will find `Some_Id` true and skip the inclusion of the file.

Note that in a project, all preprocessor identifiers start out as **false**, by default. That is, the identifiers **set true** when preprocessing a source file **do not remain true** for preprocessing the **next** source file. This is important for otherwise the header files once included will not be included in subsequent source files.

Appendix II.6. The continuation character

The purpose of continuation of character is to instruct the preprocessor to ignore the **new line** character, and consider the next line as **continuation** of the current line.

The Z++ continuation character is the escape character \. However, beyond the continuation of quoted string literals it has no other use in Z++. In particular, Z++ macros do not need the use of continuation character. **Therefore, the use of continuation character is only supported for quoted string literals**.

Appendix III. Preprocessor Macro Command

Macros in Z++ behave like functions. Z++ macros allow multiple lines rather than one single long string. Furthermore, macros can include all preprocessor commands except **include** and **macro**. In particular inside the body of a macro, arguments passed to calls to other macros can be locally or globally defined objects, or the arguments passed to the macro that is making the call to other macros.

While debugging, **break points** can be set at **any line** within the body of a **macro** with usual debugging capabilities.

The **macro** command must end with **endmacro** command. The definition of a macro, even **without parameters** requires a pair of **parentheses**. However, **no parentheses** should be used when **calling** a macro that takes no arguments. Calls to macros should **not** end with **semicolon**.

Following are a few examples of macro definitions and calls.

```
#macro withoutArguments()  // parentheses required here
      output << "In macro (withoutArguments).\n";
#endmacro

#macro singleArgument(First)
      withoutArguments    // no parentheses here
      output << "In macro (singleArgument): " << First << '\n';
#endmacro

#macro StringMacro(Value)
      output << "In macro (StringMacro): " << Value << '\n';
#endmacro

// This macro calls above macros

#macro twoArguments(First, Second)
      int check = 512;
      singleArgument (check)  // passing a local object

// Conditionals are allowed in macros

      #if nothing
            First += Second;
      #else
            First = Second + 13;
      #endif

      output << "In macro (twoArguments): " << First << '\n';

      StringMacro("A string as macro argument") // string literal
#endmacro
```

Remark. The **macro head** can be broken into **several lines**. However, everything after the **closing** parenthesis will be **ignored**, as indicated below.

```
#macro twoArguments(First,
                    Second) // everything else on this line ignored
      // start of first line of code here
#endmacro
```

Thus, you should start the **first** line of code in the body of a macro on next line.

The following executable example illustrates the points mentioned above.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //
// Global objects can be passed to macro calls.

int global = 317;

// ------------------------------------------------------------------ //

#macro triple(m, n, o)
      output << "In macro (triple): " << m << '\n';
      output << "In macro (triple): " << n << '\n';
      output << "In macro (triple): " << o << '\n';
#endmacro

// ------------------------------------------------------------------ //

#macro final(Char)
      output << "In macro (final): " << Char << '\n';
#endmacro

// ------------------------------------------------------------------ //

#macro internal(Second)
      output << "In macro (internal): " << Second << '\n';
      final('\'') // Calling macro
#endmacro

// ------------------------------------------------------------------ //

#macro inside(First)
      internal(First) // Calling macro
      output << "In macro (inside): " << First << '\n';
#endmacro

// ------------------------------------------------------------------ //
// Objects declared inside the body of a macro are global.
// Objects check and door are accessible to the main entry point.
```

```
#macro simple(First,
                  Second); // from semicolon on, all ignored

      int check = 512;

 // Calling macro, passing a global object to call.

      inside(global)

      double door = 3.5;

      triple(check, door, First) // Calling macro

      #if nothing
            First += Second;
      #else
            First = Second + 13;
      #endif

      output << "In macro (simple): " << First << '\n';

#endmacro

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      int three = 3, four = 4;

      simple(three, four) // Calling macro

// These objects were declared in macro simple.

      check = 1;
      door = 3.5;

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
In macro (internal): 317
In macro (final): '
In macro (inside): 317
In macro (triple): 512
In macro (triple): 3.5
In macro (triple): 3
In macro (simple): 17
Good-bye World!
```

Appendix IV. Standard Library

Role of language for design of complex solutions for applications, such as component, autonomous agent etc.

Ultimately, the work is done by the OS, accessible via system calls in C. For instance, sockets. Second category for library is like math functions. Third is intermediate standards, like blue tooth, which needs time to establish itself as a standard in its final form.

All these can be handled via linkage. But some are important enough to be presented as standard library for static linkage. Here we discuss the Z++ standard library, which could grow as new standards are accepted, or the use of math library becomes more necessary for applications written in Z++.

The standard library consists of several sections introduced in this appendix. Each section is packaged in a namespace, and is incorporated in a program by including its header file.

Appendix IV.1. Input and Output

Include file is: **iostream.h**.
Namespace is: **ioSpace**.

The stream library is for input from keyboard, and output to consol, and is not related to Graphical User Interface.

A single overloaded operator is used for input of numeric and string types. Similarly, a single overloaded operator is used for output of numeric and **string** types. The output also has a method to flush (buffered) output.

Appendix IV.1.A. Output

The library **class** outStream defines output.

```
class outStream
      outStream& operator=(const outStream&);
      boolean operator==(const outStream&) const;
      boolean operator!=(const outStream&) const;
public:
      outStream(void);
      ~outStream(void);
      outStream(const outStream&);
      outStream& operator<<(char);
      outStream& operator<<(uchar);
      outStream& operator<<(short);
      outStream& operator<<(ushort);
      outStream& operator<<(int);
      outStream& operator<<(uint);
      outStream& operator<<(long);
      outStream& operator<<(ulong);
      outStream& operator<<(float);
      outStream& operator<<(double);
      outStream& operator<<(string);
      outStream& flush(void);
end;
```

When the file iostream.h is included, the compiler creates the instance **output** of outStream. So, to output an object of numeric or **string** type, simply do as follows.

```
output << object;
```

You can do a sequence of outputs in a single statement. The last output below is the new line character.

```
output << "The value of object is: " << object << '\n';
```

The example at end of this section also shows how to overload output operator for other types.

Output is **buffered**. That is, characters to output are kept in a buffer until the buffer fills up, and is sent to consol. If you want to send the buffered output to consol earlier, use the flush() method.

```
output.flush();
```

Appendix IV.1.B. Input

The library **class** inStream defines input.

```
class inStream
      inStream& operator=(const inStream&);
      boolean operator==(const inStream&) const;
      boolean operator!=(const inStream&) const;
public:
      inStream(void);
      ~inStream(void);
      inStream(const inStream&);
      inStream& operator>>(char&);
      inStream& operator>>(uchar&);
      inStream& operator>>(short&);
      inStream& operator>>(ushort&);
      inStream& operator>>(int&);
      inStream& operator>>(uint&);
      inStream& operator>>(long&);
      inStream& operator>>(ulong&);
      inStream& operator>>(float&);
      inStream& operator>>(double&);
      inStream& operator>>(string&);
end;
```

When the file iostream.h is included the compiler creates the instance **input** of `inStream`.

All parameters are by reference. The result of input is stored in the argument passed to the input **operator**.

Appendix IV.1.C. Overloading IO operators

The following example shows how to overload input and output for user-defined types.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

class hidden
      int n;
```

```
      double d;
      string s;
public:
      friend outStream& operator<<(outStream&, const hidden&);
      friend inStream& operator>>(inStream&, hidden&);
end;

outStream& operator<<(outStream& o, const hidden& h)
      o << "\nInteger member is: " << h.n << '\n';
      o << "Double member is: " << h.d << '\n';
      o << "String member is: \"" << h.s << "\"\n";
      return o;
end;

// When switching from output to input, the library flushes
// the output.
// Values entered must be separated by white spaces. However,
// a string value cannot contain white spaces. For instance,
// 777 47.97 so-long
// Pressing enter is also a white space, as is tab.

inStream& operator>>(inStream& i, hidden& h)
      output << "Enter values: ";
      output.flush(); // not needed here
      i >> h.n >> h.d >> h.s;
      return i;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      hidden hdn;

      input >> hdn; // Get values from user

      output << hdn; // Show values of members

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Enter values:777 47.97 so-long
Integer member is: 777
Double member is: 47.97
String member is: "so-long"
Good-bye World!
```

Appendix IV.2. File Operations

Include file is: **stream.h**.
Namespace is: **fileSpace**.

The notion of file is an established computing abstraction, with well-understood file operations. It is possible to simulate files on handheld devices. Therefore, it is possible to provide a **single universal abstract view** of file and its related operations.

The purpose of a computing language is to provide a coherent medium for expressing solutions. Fundamental types and operations on them are general and exist independent of the capabilities of computing devices. On the other hand, the notion of file and file operations **evolved as services** of operating systems in the form of **standards**.

The notion of file is defined as a **class** type using the fundamental types. However, operations on file objects, such as opening and reading files, need the services of the **underlying** operating system (as apart from Z47 Processor). Therefore, file operations are presented as **library** function calls, which are mapped to system calls.

After presenting the library **class** FileStreamType, description of its methods and illustrations will follow.

Appendix IV.2.A. FileStreamType

The library **class** FileStreamType is defined as follows.

```
class FileStreamType
      int fileDescriptor;
      boolean state;
      uchar flags;

      FileStreamType& operator=(const FileStreamType&);
      boolean operator==(const FileStreamType&) const;
      boolean operator!=(const FileStreamType&) const;

public:

      FileStreamType(void);
      ~FileStreamType(void);
      FileStreamType(const FileStreamType&);

      boolean open(const string&);
      void close(void);
      boolean isOpen(void); // True if file is open, otherwise False
      boolean eof(void);    // True when end of file has reached

      void eatWhite(void);  // Remove and discard white spaces
      void putBack(char);   // Put the char back into the INPUT stream
```

```
        FileStreamType& operator>>(char&);
        FileStreamType& operator>>(uchar&);
        FileStreamType& operator>>(short&);
        FileStreamType& operator>>(ushort&);
        FileStreamType& operator>>(int&);
        FileStreamType& operator>>(uint&);
        FileStreamType& operator>>(long&);
        FileStreamType& operator>>(ulong&);
        FileStreamType& operator>>(float&);
        FileStreamType& operator>>(double&);
        FileStreamType& operator>>(string&);
        void read(char*, int);

        FileStreamType& operator<<(char);
        FileStreamType& operator<<(uchar);
        FileStreamType& operator<<(short);
        FileStreamType& operator<<(ushort);
        FileStreamType& operator<<(int);
        FileStreamType& operator<<(uint);
        FileStreamType& operator<<(long);
        FileStreamType& operator<<(ulong);
        FileStreamType& operator<<(float);
        FileStreamType& operator<<(double);
        FileStreamType& operator<<(string);
        void write(char*, int);

        void setInput(void);       // Make file open for input only
        void setOutput(void);      // Make file open for output only
        void setInOut(void);       // Default case, for input and output.

        void setTruncate(void);    // Truncate file to size 0 when opening
        void clearTruncate(void); // Default. Do not truncate the file.

        void setAppend(void);      // Send each output to end of file.
        void clearAppend(void);    // Default. Do not append.

        void setBinary(void);      // Make file open in binary.
        void setText(void);        // Default. Make file open in text.

    // setCurrent() is the default.

        void setCurrent(void);     // Set seek origin to current position
        void setStart(void);       // Set seek origin to start of file.
        void setEnd(void);         // Set seek origin to end of file.

        void seekInput(long);      // Do a seek on input stream.
        void seekOutput(long);     // Do a seek on output stream.

        long tellInput(void);      // Where is input at?
        long tellOutput(void);     // Where is output at?

    end;
```

Appendix IV.2.B. Opening a file

A file is identified with a filename. For opening a file pass the full path-name to open() method of **class** FileStreamType. There are several settings that you may wish to change the default values **before** invoking open(). These setting are: input/output, truncation, appending to file, binary/text.

The settings are **not flags** passed as argument to the open() method. Instead, each setting is done by, **calling** a corresponding method, discussed in the following subsections. Once a file is opened, it is **not** possible to **change** these settings. Normally, one would close the file and **re-open** it with new settings.

Note that defaults are for **initial creation** (declaration) of a stream object. In case you modify the settings, closing a stream does **not change** them. So, next time you open the same stream you may need to **adjust** the settings by calling the methods of the following subsections.

Appendix IV.2.B.1. Setting input/output direction

The default setting is to open a file for both, input from file and output to the file. There are three methods for this setting.

void setInput(**void**) : sets file for receiving input from file.

void setOutput(**void**) : sets file for sending output to file.

`void setInOut(void)` : this is the **default**. File opens for read and write.

Appendix IV.2.B.2. Truncating file

When opening an existing file, you can choose to delete all its contents before writing to the file. The default is **not** to **truncate**.

void setTruncate(**void**) : sets the size of the file to 0 upon opening the file.

void clearTruncate(**void**) : this is the **default**. File is not modified in any way.

Appendix IV.2.B.3. Appending to an existing file

When opening an existing file for writing to the file, output is sent to the start of the file. That is, file contents are overwritten. To change this default action, set appending to file before opening the file.

void setAppend(**void**) : write new data to the **end** of file.

void clearAppend(**void**) : begin writing new data from the **start** of file (**default**).

Appendix IV.2.B.4. Setting binary/text file mode

Binary files are recorded without additional characters. On the other hand, each operating system may add additional characters to the end of each line of text.

Input and output of objects of fundamental types to text files is done the same way these objects are received from keyboard or sent to the consol. However, binary read and write to files takes a little more.

The default is to open a file in text mode. The following methods deal with this setting.

void setBinary(**void**) : open file in binary mode.

void setText(**void**) : open file in text mode (**default**).

Appendix IV.2.C. Input from a file

Input from a file means reading from the file. For text files, input of fundamental types is done via the overloaded versions of **operator** >>. For binary files use the read() method.

Appendix IV.2.C.1. Input from a text file

Text files are read with overloaded versions of **operator** >>, which is overloaded for numeric and **string** types. See FileStreamType.

Appendix IV.2.C.2. Reading from a binary file

Reading a **binary** file needs a **buffer** to read the file into, via the method read().

void read(**char*** buffer, **int** length);

The parameter **length** is the number of characters to read from the file into the **buffer**.

Appendix IV.2.D. Output to a file

Output to a file means writing to the file. For text files, output of fundamental types is done via the overloaded versions of **operator** <<. For binary files use the write() method.

Appendix IV.2.D.1. Output to a text file

For writing to a **text** file use overloaded versions of **operator** <<. This **operator** is overloaded for numeric and **string** types. See FileStreamType.

Appendix IV.2.D.2. Writing to a binary file

Writing to a **binary** file needs a **buffer** for writing to the file.

void write(**char*** buffer, **int** length);

The parameter **length** is the number of characters to write from the **buffer** to the file.

Appendix IV.2.E. Moving around in a file

The operating system maintains a pointer to the next position for **reading** a character from a file, and a pointer to the next position for **writing** the next character to the file. At times we need to change these pointers to other parts of the file. This is accomplished via seek actions.

Before a seek action we can set the **origin** from which the seek will be done.

Appendix IV.2.E.1. Seek actions

`void seekInput(long distance)` : sets current position of reading from file at the **distance** from the seek **origin**.

`void seekOutput(long distance)` : sets current position of writing to file at the **distance** from the seek **origin**.

Appendix IV.2.E.2. Setting seek origin

`void setCurrent(void)` : set origin to current position maintained by operating system (**default**). Generally speaking, this method is useful as documentation in selection statements, like **if-else-endif**.

`void setStart(void)` : sets the origin of seek to the beginning of file.

`void setEnd(void)` : sets the origin of seek to the end of file. Thus, the seek action will be from end of file **inwards** towards the beginning of file.

Appendix IV.2.F. Locating position of input/output

In many file operations we need to know the position of next input/output maintained by the operating system. The **tell** methods provide that information.

`long tellInput(void)` : returns the **current** position of reading from file.

`long tellOutput(void)` : returns the **current** position of writing to file.

Appendix IV.2.G. Examples

The examples of this appendix show various ways of dealing with files. In all examples the file names must be provided before compiling the files. A file name must be the full **path-name** of the file.

When reading a **string** from a **text** file, a complete line until the **new-line** character is read into the **string** object. However, the new-line character is **left** in the stream. So, before reading the next line, use eatWhite() method to get rid of the new-line character, as well as all system related white spaces at end of line.

Appendix IV.2.G.1. Text File

This example illustrates how to overload input/output file operators when working with text files. The example reads/writes objects of fundamental types from/to text files.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<stream.h>
using namespace fileSpace;

// ------------------------------------------------------------------ //

class Values

      int n<visible>;
      long g<visible>;
      short s<visible>;
      float f<visible>;
      double d<visible>;
      char c<visible>;
      string t<visible>;

public:

      Values(void);
      void Clear(void);
friend FileStreamType& operator<<(FileStreamType&, const Values&);
friend FileStreamType& operator>>(FileStreamType&, Values&);

end;

// ------------------------------------------------------------------ //

Values::Values(void)
      n = 47;
      g = 97;
      s = 7;
      f = 4.7;
```

```
      d = 47.97;
      c = 'X';
      t = "I am a member of class Values.";
end;

void Values::Clear(void)
      n = 0;
      g = 0;
      s = 0;
      f = 0;
      d = 0;
      c = 'a';
      t = "";
end;

// ------------------------------------------------------------------ //
// Write each member on a single line of file

FileStreamType& operator<<(FileStreamType& fs, const Values& v)
      fs << v.n << '\n';
      fs << v.g << '\n';
      fs << v.s << '\n';
      fs << v.f << '\n';
      fs << v.d << '\n';
      fs << v.c << '\n';
      fs << v.t << '\n';
      return fs;
end;

FileStreamType& operator>>(FileStreamType& fs, Values& v)
      fs.eatWhite();
      fs >> v.n;
      fs.eatWhite();
      fs >> v.g;
      fs.eatWhite();
      fs >> v.s;
      fs.eatWhite();
      fs >> v.f;
      fs.eatWhite();
      fs >> v.d;
      fs.eatWhite();
      fs >> v.c;
      fs.eatWhite();
      fs >> v.t;
      return fs;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      Values Vls;

// FileName must be full path-name of file.
```

```
        string FileName = "full-path/filename";

        FileStreamType file; // Create a stream object

        file.open(FileName); // open for read/write

        file << Vls; // Write to file

        Vls.Clear(); // Clear the object

// Set input to start of file

        file.setStart(); // Set seek origin to start
        file.seekInput(0); // seek to beginning of file

        file >> Vls; // Read the file

        file.close(); // we are done

        output << "Values read back from file ... \n";
        output << "n (int == 47) : " << Vls.n << '\n';
        output << "g (long == 97) : " << Vls.g << '\n';
        output << "s (short == 7) : " << Vls.s << '\n';
        output << "f (float == 4.7) : " << Vls.f << '\n';
        output << "d (double == 47.97) : " << Vls.d << '\n';
        output << "c (char == 'X') : " << Vls.c << '\n';
        output << "t (string) : " << Vls.t << '\n';

        output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Values read back from file ...
n (int == 47) : 47
g (long == 97) : 97
s (short == 7) : 7
f (float == 4.7) : 4.7
d (double == 47.97) : 47.97
c (char == 'X') : X
t (string) : I am a member of class Values.
Good-bye World!
```

Appendix IV.2.G.2. Binary File

This example illustrates how to read/write binary values of numeric objects when working with binary files.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<stream.h>
using namespace fileSpace;

#include<char.h>
using namespace charSpace;

// ------------------------------------------------------------------ //

struct Values

      int n;
      long g;
      short s;
      float f;
      double d;
      char c;

      Values(void);
end;

Values::Values(void)
      n = 47;
      g = 97;
      s = 7;
      f = 4.7;
      d = 47.97;
      c = 'X';
end;

// ------------------------------------------------------------------ //
// Copy binary values to members of struct Values.

void copyMembers(Values& t, char* bufPtr)
      memCopy(&t.n, bufPtr, size(t.n));
      bufPtr += size(t.n);
      memCopy(&t.g, bufPtr, size(t.g));
      bufPtr += size(t.g);
      memCopy(&t.s, bufPtr, size(t.s));
      bufPtr += size(t.s);
      memCopy(&t.f, bufPtr, size(t.f));
      bufPtr += size(t.f);
      memCopy(&t.d, bufPtr, size(t.d));
      bufPtr += size(t.d);
      t.c = *bufPtr;
```

```
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      Values t;
      int length = size(t.n);
      length += size(t.g);
      length += size(t.s);
      length += size(t.f);
      length += size(t.d);
      length += size(t.c);

      output << "Length is : " << length << '\n';

// Create a buffer

      char* bufPtr = (char*) (new char[length+1]);

      char* bufHead = bufPtr;

// Copy binary values of members of struct Values to buffer

      memCopy(bufPtr, &t.n, size(t.n));
      bufPtr += size(t.n);
      memCopy(bufPtr, &t.g, size(t.g));
      bufPtr += size(t.g);
      memCopy(bufPtr, &t.s, size(t.s));
      bufPtr += size(t.s);
      memCopy(bufPtr, &t.f, size(t.f));
      bufPtr += size(t.f);
      memCopy(bufPtr, &t.d, size(t.d));
      bufPtr += size(t.d);
      *bufPtr++ = t.c;

// FileName must be full path-name of file.

      string FileName = "Full-Path-Name/filename";

      FileStreamType file; // Create a stream object
      file.setBinary(); // Make it binary

      file.open(FileName); // open for read/write

      file.write(bufHead, length); // Binary write

      memSet(bufHead, length, 0); // Clear the buffer

// Set input to start of file

      file.setStart(); // Set seek origin to start
      file.seekInput(0); // seek to beginning of file
```

```
        file.read(bufHead, length); // Binary read

        file.close(); // we are done

// Clear the object

        t.n = 0;
        t.g = 0;
        t.s = 0;
        t.f = 0;
        t.d = 0;
        t.c = 'a';

        copyMembers(t, bufHead); // Copy buffer to object

        output << "Copied values are ... \n";
        output << "t.n (int == 47) : " << t.n << '\n';
        output << "t.g (long == 97) : " << t.g << '\n';
        output << "t.s (short == 7) : " << t.s << '\n';
        output << "t.f (float == 4.7) : " << t.f << '\n';
        output << "t.d (double == 47.97) : " << t.d << '\n';
        output << "t.c (char == 'X') : " << t.c << '\n';

        output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Length is : 27
Copied values are ...
t.n (int == 47) : 47
t.g (long == 97) : 97
t.s (short == 7) : 7
t.f (float == 4.7) : 4.7
t.d (double == 47.97) : 47.97
t.c (char == 'X') : X
Good-bye World!
```

Appendix IV.2.G.3. File Read/Write

The example of this section illustrates how to open/close multiple files and set them to input/output, as well as reading from one file and writing the result to another file.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<stream.h>
using namespace fileSpace;

// ------------------------------------------------------------------ //
// FileNames must be full path-name of file.

string SourceFile = "full-path/source.txt";
string DestinationFile = "full-path/destination.txt";

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
      output << "Hello World!\n";

      string oneLine;

// Create stream objects

      FileStreamType Source;
      FileStreamType Destination;

// Write a few lines to SourceFile

      Source.setOutput(); // For write
      Source.open(SourceFile);

      oneLine = "First Line of Text.\n";
      Source << oneLine;

      oneLine = "Second Line of Text.\n";
      Source << oneLine;

      oneLine = "Third Line of Text.\n";
      Source << oneLine;

      Source.close();

// Now read from source and write to destination

      Source.setInput(); // For read
      Source.open(SourceFile);
```

```
        Destination.setOutput(); // For write
        Destination.open(DestinationFile);

        do
                Source.eatWhite();
                if (Source.eof()) break; endif;
                Source >> oneLine;
                Destination << oneLine << '\n';
        enddo;

        Source.close();
        Destination.close();

// Show what is in destination

        Destination.setInput(); // For read
        Destination.open(DestinationFile);

        while (!Destination.eof())
                Destination >> oneLine;
                output << oneLine << '\n';
                Destination.eatWhite();
        endwhile;

        Destination.close();

        output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
First Line of Text.
Second Line of Text.
Third Line of Text.
Good-bye World!
```

Appendix IV.3. Character Routines

Include file is: **char.h**.
Namespace is: **charSpace**.

These are the usual character routines of C, and a few more.

Appendix IV.3.A. isSpace

```
boolean isSpace(char c);
```

Returns **True** when c is one of: space ‘ ‘, tab ‘\t’, return ‘\r’ or new line ‘\n’.

Appendix IV.3.B. isDigit

```
boolean isDigit(char c);
```

Returns **True** when c is a digit from 0 to 9.

Appendix IV.3.C. isUpper

```
boolean isUpper(char c);
```

Returns **True** when c is an uppercase letter from ‘A’ to ‘Z’.

Appendix IV.3.D. isLower

```
boolean isLower(char c);
```

Returns **True** when c is a lowercase letter from ‘a’ to ‘z’.

Appendix IV.3.E. isAlpha

```
boolean isAlpha(char c);
```

Returns **True** when c is a letter: from ‘A’ to ‘Z’, or from ‘a’ to ‘z’.

Appendix IV.3.F. isSymbol

```
boolean isSymbol(char c);
```

Returns **True** when c is one of the following:

```
';' ':' ',' '.' '\'' '"' '#' '$' '@' '\\' '`' '_' '?'
```

Appendix IV.3.G. isOperator

```
boolean isOperator(char c);
```

Returns **True** when c is one of the following:

```
'+' '-' '*' '/' '%' '!' '~' '|' '&' '^' '?' '<' '>'
```

Appendix IV.3.H. isBracket

```
boolean isBracket(char c);
```

Returns **True** when c is one of the following:

```
'[' ']' '{' '}' '(' ')' '<' '>'
```

Appendix IV.3.I. toUpper

```
char toUpper(char c);
```

If c is a lowercase letter, returns the uppercase of c. Otherwise, returns c unchanged.

Appendix IV.3.J. toLower

```
char toLower(char c)
```

If c is an uppercase letter, returns the lowercase of c. Otherwise, returns c unchanged.

Appendix IV.4. Memory Routines

Include file is: **char.h**.
Namespace is: **charSpace**.

Appendix IV.4.A. memset

```
void memset(void* start, char value, int length);
```

Sets the value of memory locations to value, from start to start+length.

Due to promotions (implicit parameter conversion), when literal values are passed for second and third arguments, you could write wrong value to wrong location. The correct order is, value and then length.

```
// Sample.zpp

#include<iostream.h>
using namespace ioSpace;

#include<char.h>
using namespace charSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

	output << "Hello World!\n";

	char* location = (char*) new char[50];

	memset(location, 0, 50);

	memset(location, 'A', 20);

	string stg = location;

	output << stg << '\n';

	output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
AAAAAAAAAAAAAAAAAAAA
Good-bye World!
```

Appendix IV.4.B. memcpy

```
void memcpy(void* target, void* source, int length);
```

Copies bytes of memory, equal to length, from source to target.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<char.h>
using namespace charSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

	output << "Hello World!\n";

	int length = 50;

	char* source = (char*) new char[length];

	memset(source, 0, length);
	memset(source, 'B', 20);

	char* target = (char*) new char[length];

	memcpy(target, source, length);

	string stg = target;

	output << stg << '\n';

	output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
BBBBBBBBBBBBBBBBBBBB
Good-bye World!
```

Appendix IV.4.C. memcmp

```
boolean memmp(void* target, void* source, int length);
```

Returns true only if all bytes in source and target, up to length, are equal.

Appendix IV.4.D. Example

Memory routines are same as those of C. Below is an example using these routines along with pointers.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<char.h>
using namespace charSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      int length = 50;

      char* buffer = (char*) new char[length];
      memset(buffer, 0, length);

      int IntArray[5];
      int counter;

      for (counter = 0; counter < 5; counter++)
            IntArray[counter] = 2 * counter + 1;
      endfor;

      char* p;
      char* q = buffer;

// Copy values of array cells to buffer

      for (counter = 0; counter < 5; counter++)
            p = (char*) &IntArray[counter];
            memcpy(q, p, size(int));
            q += size(int);
      endfor;

// Make a small buffer.

      char* store = (char*) new char[20];
      memset(store, 0, 20);

      int holder = 0;
      q = buffer;
      counter = 0;
      char* storePointer = store;
```

```
// Show each value in buffer and copy it to store.

      do
            p = (char*) &holder;
            memcpy(p, q, size(int));
            q += size(int);
            output << "Value of holder is: " << holder << '\n';
            memcpy(storePointer, &holder, size(int));
            storePointer += size(int);
      enddo(++counter == 5);

// Make sure buffers are identical

      if (memcmp(store, buffer, 20))
            output << "Buffers are identical.\n";
      else output << "Buffers are NOT equal.\n";
      endif;

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Value of holder is: 1
Value of holder is: 3
Value of holder is: 5
Value of holder is: 7
Value of holder is: 9
Buffers are identical.
Good-bye World!
```

Appendix IV.5. String Routines

Include file is: **string.h**.
Namespace is: **stringSpace**.

Type **string** is fundamental, with direct support for concatenation of objects of **string** type, length of a **string** object etc. The library routines cover more specialized **string** operations.

The functions `strstr()` `strchr()` `getchr()` `Replace()` are 0-based as is customary in C. That is, the first character is at position 0.

Functions `Left()` `Right()` `Middle()` count the first position as 1. That is, the first character in a **string** is at position 1, not 0.

At end of this section, an example illustrates the library routines.

Appendix IV.5.A. strstr

```
int strstr(const string& Source, const string& Target);
```

If Target is a substring of Source, this function returns the index >= 0 of start of Target in Source. Otherwise, it returns –1.

Appendix IV.5.B. strchr

```
int strchr(const string& Source, char c);
```

If **char** c is found in Source, this function returns the index >= 0 of c in Source. Otherwise, it returns –1.

Appendix IV.5.C. getchr

```
char getchr(const string& Source, int index);
```

If index exceeds the length of Source, this function returns 0 (Null char). Otherwise, it returns the character at index, in Source.

Appendix IV.5.D. Left

```
string Left(const string& Source, int length);
```

This function returns a substring of size length, starting from **left** of Source.

Appendix IV.5.E. Right

```
string Right(const string& Source, int length);
```

This function returns a substring of size length, starting from **right** of Source. Note that the string returned is from left to right.

Appendix IV.5.F. Middle

```
string Middle(const string& Source, int position, int length);
```

This function returns a substring, based on values of position and length. When values of position and/or length are invalid, it returns the **empty string**. In particular, a negative value for position is invalid.

If position is 0, extraction is from right end. Otherwise, position indicates the start of extraction relative to the left end.

The value 0 for length means all the way to end of Source, in either direction.

When length is a negative number extraction is from right. That is, it starts at position into the Source, and extracts a substring of size length, going left.

Appendix IV.5.G. toUpper

```
string toUpper(const string& Source);
```

This function returns the original string with all letters in **uppercase**. All other characters remain unchanged.

Appendix IV.5.H. toLower

```
string toLower(const string& Source);
```

This function returns the original string with all letters in **lowercase**. All other characters remain unchanged.

Appendix IV.5.I. Replace

```
string Replace(const string& Given, const string& Modifier, int Start);
```

The arguments Given and Modifier are not changed. The function returns a copy of Given with part of it replaced with Modifier, beginning from Start position in Given.

The start position (first character) in Given begins at 0. Negative values, and values bigger than length of Given are invalid. When the value of Start is invalid the empty **string** is returned.

When either of the **string** arguments, Given and/or Modifier is empty, the empty **string** is returned.

If, beginning at Start position in Given, the length of Modifier crosses the end of Given, only as many characters in Modifier are copied to Given as they will fit. The remaining characters of Modifier are ignored.

Appendix IV.5.J. Reverse

```
string Reverse(const string& Argument);
```

This function returns the reverse of the **string** Argument, without changing Argument.

Appendix IV.5.K. strcpy

```
char* strcpy(char* Destination, const string& Source);
```

Copies the string Source, as C **null-terminated** character string to Destination. It returns the pointer to the start of Destination.

Appendix IV.5.L. strncpy

```
char* strncpy(char* Destination, const string& Source, int Number);
```

Copies Number (third argument) of characters from Source, as C **null-terminated** character string to Destination. It returns the pointer to the start of Destination.

Appendix IV.5.M. Examples

The following examples illustrate the library functions for manipulating strings.

```
// LeftRight.zpp

#include<iostream.h>
using namespace ioSpace;

#include<string.h>
using namespace stringSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      string Source = "I am a long string.";
      string Target = "long string.";

// Find start of a substring (0-based).

      int index = strstr(Source, Target);

// The char at index must be the letter l.

      char ch = getchr(Source, index);
      output << "Char at index: " << index << " is: " << ch << '\n';

// Left(), Right() and Middle() begin counting with 1. That is
// the first position in a string is 1, not 0.

      string left = Left(Source, index);
      output << left << '\n';

      string right = Right(Source, size(Source) - index);
      output << right << '\n';

      string Result = left + right;

      output << Result << '\n';

// Result will be "long".

      Result = Middle(Source, 8, 4);
      output << Result << '\n';

// And again, Result will be "long". When size of extraction
// is negative, the starting point (index 11) becomes the
// end of Source, and extraction is from right to left.

      Result = Middle(Source, 11, -4);
```

```
	output << Result << '\n';

	output << toUpper(Source) << '\n';

	output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Char at index: 7 is: l
I am a
long string.
I am a long string.
long
long
I AM A LONG STRING.
Good-bye World!
```

```
// RevRep.zpp

#include<iostream.h>
using namespace ioSpace;

#include<string.h>
using namespace stringSpace;

////////////////////////////////////////////////////////////////////////
//--------------------------------------------------------------------//
////////////////////////////////////////////////////////////////////////

entry void main(void)
	output << "Hello World!\n";

//--------------------------------------------------------------------//

	string operand = "This says correct";
	string replace = "ll"; // replace rr with ll
	string result = Replace(operand, replace, 12);
	output << "Result is: " << result << '\n';

//--------------------------------------------------------------------//
// If replace is too long, only the part that fits replaces operand.

	result = "initialize";
	replace = "correction";
	result = Replace(operand, replace, 10);
	output << "Result (replace too long) is: " << result << '\n';

//--------------------------------------------------------------------//

	result = Reverse(operand);
	output << "Operand is: " << operand << '\n';
	output << "Result of reversing is: " << result << '\n';
```

```
//-------------------------------------------------------------------//

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Result is: This says collect
Result (replace too long) is: This says correct
Operand is: This says correct
Result of reversing is: tcerroc syas sihT
Good-bye World!
```

```
//StrCopy.zpp

#include<iostream.h>
using namespace ioSpace;

#include<string.h>
using namespace stringSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      string source = "Long string ";
      char holder[50];

// Make a pointer to start of array holder, and
// maintain a pointer to its start.

      char* destination = (char*) holder;
      char* destPointer = destination;

      strcpy(destination, source);

      destination += size(source); // Move destination forward

// Copy new value of source after its previous value
// was copied to destination.

      source = "Short";
      string Partial = strcpy(destination, source);

// String Result will be the entire destination, but
// Partial will only be the last string added, namely "Short"

      string Result = destPointer;
```

```
        output << Partial << '\n';
        output << Result << '\n';

        destination = destPointer; // Recover start of array holder

// Copy 5 characters from right to destination

        source = "Left Middle Right";
        Result = strcpy(destination, Right(source, 5));
        output << Result << '\n';

// Copy 4 characters from left to destination.

        Result = strncpy(destination, source, 4);
        output << Result << '\n';

        output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Short
Long string Short
Right
Left
Good-bye World!
```

Appendix IV.6. Time Routines

Include file is: **time.h**.
Namespace is: **timeSpace**.

There is **one** function in this library, **overloaded** for three different parameter types. All parameters are by reference, and the function **records** its result in the parameter passed to it. The return of the function is **void**.

```
void time(long&);
```

This form sets its argument to number of seconds elapsed since 00:00:00 GMT of January 1970. It is useful for comparing elapsed time between two events.

```
void time(string&);
```

This form sets its argument to an ASCII string for current time. The string looks something like the following.

```
      Wed Dec 19 16:35:41 2012
```

```
void time(localTimeType&);
```

This form fills in the members of argument passed to it. The type `localTimeType` is defined as follows.

```
struct localTimeType
      int seconds;
      int minutes;
      int hours;
      int month;
      int year;
      int dayOfWeek; ///////// Sunday == 0 (0 - 6)
      int dayOfMonth; //////// January == 0 (0 - 11)
      int dayOfYear;  //////// (0 - 365)
      int isDayLightSaving; // none-zero if daylight saving in effect
end;
```

Generally, C library uses year 1900 as the origin. That is, the member year of structure `localTimeType` for year 2012 will be set to 112. Z++ sets this member to **current** year.

Appendix IV.6.A. Example

The following example uses all three forms of library function time().

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

#include<time.h>
using namespace timeSpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      long startTime, endTime;
      time(startTime); // Start of execution

      output << "Hello World!\n";

      string Today;

      time(Today);
      output << Today << '\n';

      localTimeType lTT;
      time(lTT);

      output << "Values of localTimeType members are:\n";

      output << "seconds: " << lTT.seconds << '\n';
      output << "minutes: " << lTT.minutes << '\n';
      output << "hours: " << lTT.hours << '\n';
      output << "month: " << lTT.month << '\n';
      output << "year: " << lTT.year << '\n';
      output << "dayOfWeek: " << lTT.dayOfWeek << '\n';
      output << "dayOfMonth: " << lTT.dayOfMonth << '\n';
      output << "dayOfYear: " << lTT.dayOfYear << '\n';
      output << "isDayLightSaving: " << lTT.isDayLightSaving << '\n';

      output << "Good-bye World!\n";

      time(endTime); // end of execution
      output << "\nElapsed time in seconds: " <<
                  endTime - startTime << '\n';

end;
```

The output of this example is as follows.

```
Hello World!
```

```
Wed Dec 19 16:35:41 2012

Values of localTimeType members are:
seconds: 41
minutes: 35
hours: 16
month: 11
year: 2012
dayOfWeek: 3
dayOfMonth: 19
dayOfYear: 353
isDayLightSaving: 0
Good-bye World!

Elapsed time in seconds: 0
```

Appendix IV.7. Socket Library

Include file is: **transmit.h**.
Namespace is: **transmitSpace**.

This library includes **class** transmissionStreamType. The type overloads socket input and output analogous to ordinary input/output. Other **class** types can overload these operators as is done with iostream.

Appendix IV.7.A. Constructor

The only **public** constructor is transmissionStreamType(**uint**). The argument to the constructor is an opened socket, usually coming from client or server.

Appendix IV.7.B. Input

Operator >> is overloaded for fundamental types: **char, uchar, short, ushort, int, uint, long, ulong, float, double** and **string**. Below is the prototype of this **operator** for **string**.

```
transmissionStreamType& operator>>(string&);
```

Appendix IV.7.C. Output

Operator << is overloaded for fundamental types: **char, uchar, short, ushort, int, uint, long, ulong, float, double** and **string**. Below is the prototype of this **operator** for **string**.

```
transmissionStreamType& operator<<(string);
```

Appendix IV.7.D. Send

Method send, takes a buffer and an **int** for the size of the buffer, and sends it over the socket. The prototype is as follows.

```
void send(char*, int);
```

Appendix IV.7.E. Receive

Method receive, takes a buffer and an **int** for the size of the buffer, and attempts to receive as many characters as size. The prototype is as follows.

```
void receive(char*, int);
```

Appendix IV.7.E. Example

For an example with using packing see Disengage Signals. However, sending more than two pieces of data should use packing in order to reduce the size of communication over the Internet. The example of this appendix is a rewriting of the above-mentioned example, using packing.

```
//Packing.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////////////////////////////////////////
// Important. Before compiling please set the ip address for
// creating client to the IP address of machine that is running
// the Server, which could be the same machine.
// The function is: client(void).
// ------------------------------------------------------------------ //
// Make sure port you are using is open. Here we are using 5001.
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// Include header file for Internet transmission
// (client/server, sockets).
// ------------------------------------------------------------------ //

#include<transmit.h>
using namespace transmitSpace;

////////////////////////////////////////////////////////////////////////
// Make some events by extending system events.
// ------------------------------------------------------------------ //

enum MyServiceSignals : signalEventType {
      _SIGNAL_TerminateServer,
      _SIGNAL_ClientCompleted,
      _SIGNAL_ServerInitializationComplete,
      _SIGNAL_ServerTerminationComplete
};

////////////////////////////////////////////////////////////////////////
// Define a type of global thread that will be used to process
// client requests. The name of type is requestProcessorType.
// ------------------------------------------------------------------ //

type void requestProcessorType(uint)<thread>;

typedef DegenArrayType char[];

////////////////////////////////////////////////////////////////////////
// In order to show how to extend packing/unpacking to members of
```

```
// a class, first we define a member for send/receive over socket.
// ------------------------------------------------------------------ //

struct transmissionMemberType
      float f;

      transmissionMemberType(float) cast;
      int pack(char*, transmissionStreamType&, int) const;
      int unpack(char*, transmissionStreamType&, int) const;
      int Length(void) const;

      friend outStream& operator<<(outStream&,
            const transmissionMemberType&);
end;

transmissionMemberType::transmissionMemberType(float fa)
      f = fa;
end;

int transmissionMemberType::pack(char* buffer,
            transmissionStreamType& streamInstance, int index)

      return streamInstance.pack(buffer, f, index);
end;

int transmissionMemberType::unpack(char* buffer,
            transmissionStreamType& streamInstance, int index)

      return streamInstance.unpack(buffer, f, index);
end;

// What is the size of this object

int transmissionMemberType::Length(void)
      return size(float);
end;

outStream& operator<<(outStream& streamInstance,
            const transmissionMemberType& object)

      streamInstance << object.f << '\n';
      return streamInstance;
end;

////////////////////////////////////////////////////////////////////
// Here we define the type of object that we will use for sending
// from server to client.
// ------------------------------------------------------------------ //

struct transmissionObjectType
      string g;
      char c;
      short s;
      int i;
      double d;
      transmissionMemberType tmt;
```

```
      transmissionObjectType(const string&, char, short,
            int, double, float);

      friend transmissionStreamType& operator<<(
                  transmissionStreamType&,
                  consttransmissionObjectType&);

      friend transmissionStreamType& operator>>(
                  transmissionStreamType&, transmissionObjectType&);

      friend outStream& operator<<(
                  outStream&, const transmissionObjectType&);
end;

// ------------------------------------------------------------------ //

transmissionObjectType::transmissionObjectType(const string& ga,
            char ca, short sa, int ia, double da, float af)
: tmt(af)

      g = ga;
      c = ca;
      s = sa;
      i = ia;
      d = da;
end;

// ------------------------------------------------------------------ //
// For sending objects through socket connection. We pack the data
// in order to reduce socket communication.
// ------------------------------------------------------------------ //

transmissionStreamType& operator<<(transmissionStreamType&
            streamInstance,
            const transmissionObjectType& object)

// Calculate the entire size data to send. Add 1 to length of each
// string for null-terminator.

      int index = size(object.g) + 1 + size(object.c) +
                  size(object.s) + size(object.i) +
                        size(object.d) + object.tmt.Length();

// Make a dynamic array of chars of calculated size.

      DegenArrayType* dcptr = new char[index];

// Cast the array to char* to pass to packing methods.

      char* buffer = (char*) dcptr;

// Always start at 0 for packing.

      index = 0;
      index = streamInstance.pack(buffer, object.g, index);
      index = streamInstance.pack(buffer, object.c, index);
      index = streamInstance.pack(buffer, object.s, index);
```

```
      index = streamInstance.pack(buffer, object.i, index);
      index = streamInstance.pack(buffer, object.d, index);
      index = object.tmt.pack(buffer, streamInstance, index);

      streamInstance << index; // First send size of data
      streamInstance.send(buffer, index); // Now send the data

// Delete pointer to dynamic array, not what it was cast to.

      delete dcptr;

      return streamInstance;
end;

// ------------------------------------------------------------------ //
// For receiving objects via socket connection
// ------------------------------------------------------------------ //

transmissionStreamType& operator>>(transmissionStreamType&
            streamInstance,
            transmissionObjectType& object)


      int index;

// First get the size to receive

      streamInstance.operator>>(index);

// Make a dynamic array of chars for the size we received.

      DegenArrayType* dcptr = new char[index];

// Cast the array to char* to pass to unpacking methods.

      char* buffer = (char*) dcptr;

      streamInstance.receive(buffer, index); // Receive data

      index = 0; // Always start at 0, and unpack

      index = streamInstance.unpack(buffer, object.g, index);
      index = streamInstance.unpack(buffer, object.c, index);
      index = streamInstance.unpack(buffer, object.s, index);
      index = streamInstance.unpack(buffer, object.i, index);
      index = streamInstance.unpack(buffer, object.d, index);
      index = object.tmt.unpack(buffer, streamInstance, index);

// Delete pointer to dynamic array, not what it was cast to.

      delete dcptr;

      return streamInstance;
end;

// ------------------------------------------------------------------ //
// Plain output streaming
```

```
// ------------------------------------------------------------------ //

outStream& operator<<(outStream& streamInstance,
            const transmissionObjectType& object)

      streamInstance << object.g << '\n';
      streamInstance << object.c << '\n';
      streamInstance << object.s << '\n';
      streamInstance << object.i << '\n';
      streamInstance << object.d << '\n';
      streamInstance << object.tmt << '\n';

      return streamInstance;
end;

//////////////////////////////////////////////////////////////////////
// Derive your server class from library class serverType.
// ------------------------------------------------------------------ //
// To your server class, add a new method to serve client requests.
// In this example we have added "void service(requestProcessorType)",
// which takes one argument of type 'requestProcessorType'
// defined earlier.
// ------------------------------------------------------------------ //

class MyServer : serverType

public:

      void service(requestProcessorType); // user-defined
end;

// ------------------------------------------------------------------ //
// When method service() receives an argument for its parameter worker,
// it passes the member transDescriptor to it. TransDescriptor
// is mapped to a socket. The argument for worker is a thread
// (defined below). So, the call begins a thread to respond
// to the connection accepted by server.
// ------------------------------------------------------------------ //

void MyServer::service(requestProcessorType worker)
      worker(transDescriptor);
end;

// ------------------------------------------------------------------ //
// This is the global thread function that is passed
// to MyServer::service().
// ------------------------------------------------------------------ //

void processRequest(uint descriptor)<thread>

// Make an instance of library class transmissionStreamType for socket
// transmission. descriptor is the connection socket.

      transmissionStreamType tst(descriptor);

// Make an object for sending to client
```

```
      string stg = "Server sent following: ";
      transmissionObjectType tot(stg, 'G', 7, 47, 97.47, 47.7);

// and send it through socket

      tst << tot;

end;

////////////////////////////////////////////////////////////////////////
// This is a user-defined global thread for server's accepting
// connections and invoking a service thread to respond to the
// client's connect call.
// ------------------------------------------------------------------ //
// The specification of (comma-separated) signals following the colon
// after the term 'thread' is with regard to the accept() call.
// The Z++ processor will check for such signals even when the thread
// is blocked on accept() and will disengage when the thread
// receives such a signal. The list of signals is therefore called,
// the disengage list.
////////////////////////////////////////////////////////////////////////

void serviceLoop(void)<thread : _SIGNAL_TerminateServer>

// We have defined the function type requestProcessorType, and a
// global thread processRequest with same signature. Here, we
// make an instance to pass to the method service (its worker
// parameter is the recipient).

      requestProcessorType reqProc = processRequest;

// Make an instance of our server

      MyServer Server;

// Use port number to initialize server

      Server.initialize(5001);

// Tell caller (in this case main) that server initialization
// is complete

      signal <- _SIGNAL_ServerInitializationComplete;

// Now wait for an incoming connect request

      do
            Server.accept();

// If told to terminate, break out of the loop

            if (signal ? _SIGNAL_TerminateServer) break; endif;

// Otherwise make a thread to respond to client, and go back to waiting

            Server.service(reqProc);
```

```
      enddo;

// Tell caller that server has terminated

      signal <- _SIGNAL_ServerTerminationComplete;
end;

////////////////////////////////////////////////////////////////////////
// Let us also make a global thread for the client,
// though not necessary for this simple example.
// ------------------------------------------------------------------ //

void client(void)<thread>

// Make an instance of library class clientType.
// Set IP to Server machine

      clientType Client("192.168.0.2", 5001);

// Establish connection with server

      Client.connect();

// Make an instance of library class transmissionStreamType to receive
// something from server. getDescriptor() returns the
// connection socket.

      transmissionStreamType tst(Client.getDescriptor());

// Create an object for receiving

      transmissionObjectType tot;

// Get the object through socket

      tst >> tot;

      output << "Client got :\n\n" << tot;

// Tell caller, main(), we are done

      signal <- _SIGNAL_ClientCompleted;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

// Begin the server

      serviceLoop();

// Wait until server tells us that it is ready for
```

```
// accepting connections

      do
      enddo(signal ? _SIGNAL_ServerInitializationComplete);

// Now begin a client

      client();

// Wait until client is done

      do
      enddo(signal ? _SIGNAL_ClientCompleted);

// Now tell server to terminate

      signal <- _SIGNAL_TerminateServer;

// Wait until server tells us that it has terminated

      do
      enddo(signal ? _SIGNAL_ServerTerminationComplete);

// We are all done.

      output << "Good-bye World!\n";
end;
```

The output of this example is as follows.

```
Hello World!
Client got :

Server sent following:
G
7
47
97.47
47.7

Good-bye World!
```

Appendix IV.8. Client Server Model

Include file is: **transmit.h**.
Namespace is: **transmitSpace**.

This library presents an object-oriented model for client/server. The library automates several, at times confusing routines. You derive your client and server from library classes, which do most of the work.

Appendix IV.8.A. Server class

Library Server is defined by **class** serverType.

```
class serverType
      int serverID;
      int port; // connection port
      uchar flags;

protected:

      uint transDescriptor; // connection socket

      serverType(const serverType&);
      serverType& operator=(const serverType&);
      boolean operator==(const serverType&) const;
      boolean operator!=(const serverType&) const;

public:

      serverType(void);
      ~serverType(void);
      void initialize(int);
      void accept(void);
end;
```

Members `serverID` and `flags` are for library internal use.

Appendix IV.8.B. Server Initialization

User derives a **class** from library **class** serverType for use in an application. The derived **class**, call it Server, is started by calling the method initialize() on it. The initialize() method needs a port number for Server to listen to. However, at this point only Server initialization is completed, but the listening does not start, yet.

Appendix IV.8.C. Server accept wait-loop

The instance of derived **class** Server begins **listening** to its assigned **port** number when the method accept() is called on it. At this point, Server is live and ready to respond to incoming requests from clients.

Appendix IV.8.D. The Service Method

The derived **class** Server needs a user-defined method for responding to incoming requests from clients. The service method should use a (global) **thread** for responding to incoming requests.

Appendix IV.8.E. Server Termination

For server termination use the Z++ abstraction discussed in Disengage Signals, where you will also find an example illustrating the use of **class** library serverType. The connection socket is closed by the destructor.

Appendix IV.8.F. Client class

```
class clientType
      string serverIP;
      int port; // connection port
      uchar flags;

protected:

      uint transDescriptor; // connection socket

      clientType(const clientType&);
      clientType& operator=(const clientType&);
      boolean operator==(const clientType&) const;
      boolean operator!=(const clientType&) const;

public:

      clientType(void);
      ~clientType(void);
      clientType(string, int); // server IP and port
      void connect(void);
      uint getDescriptor(void);
end;
```

Member flags is for library internal use. Members serverIP and port are assigned by, the constructor at time of creation of client. Member transDescriptor is the connection socket.

Appendix IV.8.G. Client Initialization

Usually, there is no need to derive from library **class** clientType. One creates an instance of the library **class** using the constructor `clientType(string, int)` and passing the IP of server and the connection port to it. At this point the client does not attempt to connect to server.

Appendix IV.8.H. Connecting to Server

The client connects to the server via its method `connect(void)`. After this the client can communicate with server.

Appendix IV.8.I. The Connection Socket

The connection socket is needed for communication with server. Since generally we are not dealing with a derived **class**, as is the case with library **class** serverType, instead we can use the method `getDescriptor(void)` to get the connection socket. For an example please see Disengage Signals.

Appendix IV.8.J. Client Termination

The connection socket is closed by the destructor.

Appendix IV.9. Database Interface

Include file is: **database.h**.
Namespace is: **databaseSpace**.

Database Interface consists of two parts. The **linguistic** interface is an extended form of Structured Query Language (SQL), which allows mixing Z++ **objects** and expressions with database notions of tables, records and fields. The **library** interface comprises of two parts. The **visible** part is the construction of database objects via declarations of library classes. However, the methods of these classes are used indirectly. The compiler **translates** SQL statements to **calls to the methods** of library classes.

The above pattern of adding library increases the expressiveness of language. The compiler deals with linguistic abstractions translating them to calls that are serviced by other systems. This pattern is different from how C makes system calls, and the compiler is entirely left out because the language does not have the abstractions for compiler's participation. However, there is a difference between system programming where C is the best choice, and user-application development.

It is possible to establish **multiple** database sessions with same, and/or different database servers. The number of connections with a particular DBMS using the **same user-name** depends on the capabilities of the DBMS.

The kinds of databases supported in a Z++ version are specified in the following enumeration type.

```
enum systemDatabaseKinds {
      _Database_Kind_Mysql = 0,
      _Database_Kind_Ingres,
      _Database_Kind_Postgres,
      _Database_Kind_Proxy_Client = 62,
      _Database_Kind_Proxy_Server = 63,
      _Database_Kind_Invalid = 64
};
```

The kinds starting at 62 and up are for internal use and will be discussed later. In the above definition database systems **MySql**, **Ingres** and **Postgres** are supported.

The design of Z++ is such that **any** DBMS that supports SQL can easily be added. The important part is that Z++ database statements **remain the same** for all such additions in the **future**.

Z++ database statements are for interaction with an **existing** database. That is, only the Data Manipulation Language (DML) subset of SQL is supported. Normally, one uses specialized tools for interacting with a DBMS for creating new databases and views. Thus, Z++ linguistic interface is not intended for database administration.

Appendix IV.9.A. Library Interface

Database interaction begins with the operations: Connect to a database, log in user, and establish a session. Eventually, session must be closed, user logged off and connection to the database closed. We discuss these in the following subsections.

Appendix IV.9.A.1. Connecting to a Database

The **constructor** of **struct** databaseType is responsible for **connecting** to a database.

```
struct databaseType
      int descriptor;
      systemDatabaseKinds kind;
      string host;
      int port;
      string database;
      string connection;
      int limit;
      string proxy;

      databaseType(string hst = "",
            int prt = 0,
            string db = "",
            string cnct = "",
            systemDatabaseKinds k = _Database_Kind_Invalid,
            int lim = 0,
            string prxy = "local"
            );

end;
```

The only use of databaseType in Z++ is a **declaration**. You should **not directly** access the members of this structure. The meanings of members are as follows.

host: this is the IP address of database server.
port: port number for connecting to database server.
database: the specific database you wish to interact with.

descriptor: the session number for internal use, only.

connection: this is a placeholder in case a particular server expects additional information for accepting connection request.

kind: the kind of DB management system, such as MySQL, Ingress, etc.
limit: size of fetch, where 0 means fetch all.
proxy: in case we are going through a proxy, this is the IP address of proxy.

In general, you will only need to know the values for the first three parameters, and the kind of database.

The size of parameter for member **limit** determines the size of fetch for the SQL **select** statement. The value 0 means unlimited, equivalently, all records that satisfy the conditions of the select statement. For any other value for **limit**, after executing select, you will get the number of records indicated in limit. Then, you can repeatedly continue to get the same number until there are no more records to fetch, or you cancel fetching the remaining records. This is generally useful for **not** overwhelming handheld devices.

A typical declaration may look like the following. The database object being created is Dbase, which we will use in the next section for establishing a session.

```
databaseType Dbase(IP-address, port-no, db-name, db-kind);
```

Appendix IV.9.A.2. Establishing a Session

After connecting to a database server, you will need to establish a session. The **constructor** of the library **class** databaseUserType does that.

```
class databaseUserType
      databaseType database;
      string user;
      string password;

      void executeSelect(string&);
      void obtainResult(void);
      boolean fetchNextRow(void);
      void Mapper(string&, short, short);

      void setValueField(string&, short);
      void setValueSet(string&, short);
      void executeInsert(string&);
      void executeRemove(string&);
      void executeUpdate(string&);
      void executeFree(void);

public:

      databaseUserType(const databaseType&, string, string pw = "");
      ~databaseUserType(void);

      databaseUserType(void);
      databaseUserType(const databaseUserType&);
      databaseUserType& operator=(const databaseUserType&);

end;
```

For the **constructor** of this **class** you will need to know the **user-id**. The password is only needed in case you use a password to gain access to your intended database.

Typically, after connecting to a database server, you will establish a session as follows.

```
databaseUserType Duser(Dbase, user-name, password);
```

The session object is Duser, and the first argument Dbase is the object created in the previous section. That is, a database connection and session are typically established by making the following two declarations. The Dbase object created on first line is passed as the first argument to the second declaration.

```
databaseType Dbase(IP-address, port-no, db-name, db-kind);
databaseUserType Duser(Dbase, user-name, password);
```

The session object Duser is used as an argument in Z++ database statements.

Appendix IV.9.A.3. Ending a database Session

The session is closed when the **session** object Duser goes out of scope. The database **connection** is closed when the object Dbase goes out of scope. In both cases the destructors do the cleanup.

Appendix IV.9.B. Language Interface

Z++ database statements are extended forms of SQL. The extensions allow using **Z++ objects and expressions in SQL statements**.

Z++ extensions use the notion of **catalyst**, which we explain here. The design is intended for large sets of data. Thus, even when you wish to insert a single row in a table, you will need to use a container, like a List **template**, and the row to be inserted will have to be an element in the container (an instance of **catalyst**). But the pattern is simple, and in most cases there are more than one single row to deal with.

A **catalyst** is just a Z++ **struct** or **class**. However, you **view** a **catalyst** as a representation of a record (row) of the table in a database that you wish to interact with. The **catalyst** does not need to represent the entire record, but only those fields (columns) of the record that you are interested in. That means, a **catalyst** can, and usually does have **fewer** members than the fields of the record it represents.

In order to explain the role of a **catalyst**, suppose you are interacting with a table or view in the database called MyTable. Suppose the following three fields are among the fields of MyTable.

Name : **string**
Age : **short**
Gender : **character**

The **catalyst** for a record with the above fields would be something like the following.

```
struct record
      string name;
      short age;
      char sex;
end;
```

The **catalyst** is just a **struct** or a **class** and can have **other members** and whatever methods as needed. In Z++ SQL statements a simple mapping **associates** a **class** to a **table** thereby turning the **class** into a **catalyst**. For our example, the mapping takes the following form.

```
MyTable.Name<name>, MyTable.Age<age>, MyTable.Gender<sex>
```

The above mapping associates fields of MyTable to members of **struct** record, which makes the **struct** record a **catalyst** for the SQL statement in which it is used.

The use of **catalyst** will be illustrated in the following subsections. Here we only intended to introduce the notion. However, we will continue to use the example of this section in our illustrations.

Appendix IV.9.B.1. SQL select statement

The SQL SELECT is called **databaseSelect** in Z++. All variations of SQL **select** are supported with a **slightly extended syntax** for incorporating the **catalyst** and other objects into the statement.

Suppose we want to make a simple query and retrieve all rows of MyTable. The fields we are interested in are Name, Age and Gender. Obviously we need a **container** to store the result of the query. Let us use the List **template** for our container.

First, we instantiate the List **template** with **struct** record.

```
List<record> recordList;
```

Next we need to choose which method of List to be used for populating the container. Suppose we choose the method append() of List to insert items into recordList.

With these preparations, the **select** statement takes the following form. Note that Duser is the session object we created in Establishing a Session.

```
databaseSelect<Duser, record : MyTable>
      MyTable.Name<name>,
      MyTable.Age<age>, MyTable.Gender<sex>
      using recordList, append;
```

The first set of arguments to **databaseSelect** consists of session object, **catalyst** and tables. After the colon there can be several comma-separated tables. When there is more than one table **Z47 Processor** will perform a **join**.

The second set is a list of mappings as we discussed earlier. Mapping means that fields of each row of MyTable, after retrieval, are copied to corresponding members of the **struct** record. The **correspondence** is the mapping specified in the statement.

The last line tells the compiler, after mapping fields of a retrieved row to the **struct** record, to insert the record into the container recordList using its method append().

Earlier we mentioned that when the parameter limit is set to 0, the above **databaseSelect** statement would retrieve **all** the rows of MyTable and insert them into recordList. This means that after the execution of the statement, the container recordList will have the entire **result** of the **query**.

Appendix IV.9.B.2. SQL conditional clauses

All Z++ SQL statements accept conditional clauses, like **when** and **where**, the same way as regular SQL statements. Below is the select statement of previous section using a **where** clause.

In a conditional clause you can use either the field name (name of column) by itself, as it is usually done, or qualify the field name with its table. In the example below, we are only using the field name Age. However, you can also use `MyTable.Age`.

Remark. Do **not** enclose table-names and field-names between quotes.

```
databaseSelect<Duser, record : MyTable>
      MyTable.Name<name>,
      MyTable.Age<age>, MyTable.Gender<sex>
      where (Age < 35) // MyTable.Age < 35 is fine too
      using recordList, append;
```

Remark. DBMS are usually **case sensitive** relative to names of tables and columns. Furthermore, some servers, like Postgres do not seem to accept table-name.field-name with any form of enclosing in double quotes. At this time, for uniformity of where and set expressions, Z++ compiler only sends the name of the column to the server, whether or not you use the qualified format "table-name.field-name".

Appendix IV.9.B.3. Z++ expressions in SQL conditions

A conditional clause can also include programming **expressions**. The compiler evaluates expressions and constructs an ordinary clause before sending it to the DBMS. The example below is the select statement of previous section using a **Z++ expression** instead of literal value of 35.

```
int ageLimit = 10;
databaseSelect<Duser, record : MyTable>
      MyTable.Name<name>,
      MyTable.Age<age>, MyTable.Gender<sex>
      where (Age < {3 * ageLimit + 5})
      using recordList, append;
```

Z++ expressions appearing in a conditional clause **must be enclosed between {}** so the compiler knows it is dealing with a **programming expression**.

Appendix IV.9.B.4. SQL insert statement

In this section we continue to use the running example. Suppose we have inserted several items in the list `recordList` and we wish to insert all of them into table `MyTable`. We need to declare an iterator for use by the **`databaseInsert`** statement.

```
Iterator<record> ITR;
for (ITR = recordList; ITR; ITR++)
      databaseInsert<Duser, record : MyTable>
            MyTable.Name<name>,
            MyTable.Age<age>,
            MyTable.Gender<sex>
            using ITR.Current();
endfor;
```

The above will iterate the list `elementList` and on each iteration the item pointed to by `ITR.Current()` will be inserted into the table `MyTable`. The parts of **databaseInsert** are same as **databaseSelect**.

Appendix IV.9.B.5. SQL delete statement

Using our running example let us delete a few items from the table `MyTable`.

```
for (ITR = recordList; ITR; ITR++)

      if (ITR.Current().name == "Tarzan" ||
            ITR.Current().name == "Jane")

            databaseRemove<Duser, record : MyTable>
                  MyTable.Name<name>
                  where (Name == {ITR.Current().name});

      endif;
endfor;
```

Notice that the Z++ expression in the where-clause is enclosed between {}, as discussed in Section B.3 of this appendix.

The statement will iterate the list `recordList` and will **remove** two items from the table `MyTable`, if they are in the table.

Appendix IV.9.B.6. SQL update statement

Modifying records (rows) of a table is just as simple. Consider the following statement.

```
for (ITR = recordList; ITR; ITR++)

      if (ITR.Current().name == "John")

            databaseUpdate<Duser, record : MyTable>
                  MyTable.Name<name>,
                  MyTable.Age<age>

                  set Name = "Jack", Age = 55
                  where (Name == {ITR.Current().name});

            break; // leave the loop

      endif;
endfor;
```

This will iterate the list `recordList`. If it finds an item with name == "John" in the list, it will change the row in table `MyTable` with that Name to "Jack" and set the Age to 55.

Appendix IV.9.B.7. SQL fetch statement

The fetch statement is quite useful for getting the result of a select statement in small pieces, especially for handheld devices. The number of rows resulting from a select statement could be very large. In Section 9.A.1. Connecting to Server, we mentioned that in order to limit the size of each fetch, a value larger than 0 (default) should be passed, as shown below.

```
int fetch-size = 10;
// other arguments set here
databaseType Dbase(IP-address, port-no, db-name, db-kind, fetch-size);
```

When this is done, the select statement will only fetch 10 rows (in this example). After processing them, we need to call the fetch repeatedly to get 10 rows, until there is no more result.

Fetch statement is identical to its corresponding select statement except it does not use conditional phrases, like where.

The fetch statement returns a **boolean**. When there are still more rows to fetch the returned value will be **true**. Otherwise, if the last fetch actually retrieved all remaining rows the value returned will be **false**. For instance, if fetch-size is 10, and there are only 10 or fewer remaining row, fetch will **retrieve** them, **and also** return **false**.

After fetch returns false, calling it again will simply return false without doing anything. Below is an example for using fetch.

```
boolean MoreToFetch = True;

databaseSelect<Duser, record : MyTable>
      MyTable.Name<name>,
      MyTable.Age<age>, MyTable.Gender<sex>
      where (Age < 35)
      using recordList, append;

do

// Process the rows in recordList.
// Make sure recordList is emptied before next fetch.

      if (!MoreToFetch) break; endif; // We are done

      MoreToFetch = databaseFetch<Duser, record : MyTable>
            MyTable.Name<name>,
            MyTable.Age<age>, MyTable.Gender<sex>
            using recordList, append;

enddo;
```

Note that fetch only makes sense after a select statement. In addition, if the size of limit for fetch is 0 (the default value), select will fetch the entire result, and there is no need for calling fetch after the select statement.

Appendix IV.9.B.8. SQL free statement

The free statement should **only** be called when the fetch statement is used. The purpose of free statement is to discard the remaining rows resulting from a select statement. That is, after select we repeatedly call fetch as explained in previous section to get more rows. At some point we may decide to stop fetching the remaining rows. In this situation we need to tell the server that we are not interested in the buffered result, so the server will perform a clean up.

In the pattern shown in previous section we do **not need** to call free because the last fetch which returns false, will also do the clean up. However, it is harmless to call free after fetching is completed. On the other hand, free **should not** be called randomly, for instance after an insert or update.

The free statement is quite simple, as shown below.

```
databaseFree<Duser>;
```

That is, the free statement only needs the session object `Duser`.

Appendix IV.9.C. A complete example

The following is an actual example that has been compiled and executed, with output. It is written for interaction with a remote Ingres DBMS. It includes exception handling and can be used as a sample for writing your own programs.

```
// Example.zpp

/////////////////////////////////////////////////////////////////////
// This example illustrates the use of Z++ SQL statements for
// insert, update and remove. It also illustrates the use of
// database exceptions along with cases of resumption.
// The SQL select is also used.
// ------------------------------------------------------------- //
// This example uses database named "demodb", and table
// named "children". The DBMS for this example is a remote
// Ingres Server. Initially, there are four records (rows)
// in the table.
// ------------------------------------------------------------- //
// Before running this program, you will need to create the
// table "children" with four fields (columns):
//           name, last, age, gender.
// The fields name and last are variable chars, age is short int,
// and gender is char.
// ------------------------------------------------------------- //
// For Ingres, you also need to create a vnode (virtual node).
/////////////////////////////////////////////////////////////////////

#include<Iterator.h>

#include<iostream.h>
using namespace ioSpace;

#include<database.h>
using namespace databaseSpace;

#include<exception.h>
using namespace exceptionSpace;

// ------------------------------------------------------------- //
// The following values were used for this example.
// The user-name, password and the virtual node must be set
// after you have created them.
// ------------------------------------------------------------- //

string dbServerIP = "192.168.1.64"; // IP address of Server
int dbPort = 21064;                 // port number
string databaseName = "demodb";     // Database name
string userName = "UserName";
string userPassWD = "PassWord";     // password if needed
string vnode = "Remote";            // name of virtual node

// ------------------------------------------------------------- //
```

```
// We use this structure for mapping the four fields of the
// table "children" to members of a Z++ object.
// You can define any class with methods to do whatever you like.
// This is only an illustration.
// ------------------------------------------------------------- //

struct element
      string first;
      string last;
      short age;
      char sex;
end;

/////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------- //
// This is the start of the entry point.
// Exception _EXCEPTION_EmptyContainer could be thrown by
// template library. Z++ compiler requires the specification
// of thrown exceptions. That is, any exception thrown by a call
// in the body of a function, must appear in its throw list.
// ------------------------------------------------------------- //
/////////////////////////////////////////////////////////////////////

entry void main(void) throws(_EXCEPTION_EmptyContainer)

// Open namespace for List template and instantiate the template
// with type element (the catalyst).

      using namespace ztlListIteratorSpace;
      List<element> elementList;

      boolean executionFailure = False;

// This is the start of outermost exception layer. There are
// exception layers inside this layer, with resumption from failure.

      layer<exceptionEventType>

// Connect to Server, and establish a session. The session object
// is Duser.

            databaseType Dbase(dbServerIP, dbPort,
                  databaseName, vnode, _Database_Kind_Ingres);

            databaseUserType Duser(Dbase, userName, userPassWD);

// Create an iterator for traversing elementList

            Iterator<element> ITR;

// ------------------------------------------------------------- //
// Insert a few rows in the table "children".
// First put the objects in the container elementList. We will
// then iterate the container and insert its contents into table.
// ------------------------------------------------------------- //

            element et;
```

```
            et.first = "Tarzan";
            et.last = "Jungle";
            et.age = 33;
            et.sex = 'M';
            elementList << et;

            et.first = "Jane";
            et.last = "Forest";
            et.age = 23;
            et.sex = 'F';
            elementList << et;

            et.first = "Cheeta";
            et.last = "Tree";
            et.age = 3;
            et.sex = 'M';
            elementList << et;

// ------------------------------------------------------------------ //
// Now loop over the container and insert the objects in the table.
// If an insertion fails, skip it and continue with insertion
// by doing a resume.
// ------------------------------------------------------------------ //

            for (ITR = elementList; ITR; ITR++)

                  layer<exceptionEventType>

                        databaseInsert<Duser, element : children>
                              children.name<first>,
                              children.last<last>,
                              children.age<age>,
                              children.gender<sex>
                              using ITR.Current();

                  handler

                        case _EXCEPTION_DATABASE_InsertRequestFailed:
                              output << "Insertion of " <<
                                    ITR.Current().first << " failed\n";

                              resume;

                  endlayer;

            endfor;

// empty the container before select populates it

            -elementList;

            output << "After insertion ...\n\n";

            layer<exceptionEventType>

                  databaseSelect<Duser, element : children>
```

```
                        children.name<first>,
                        children.last<last>,
                        children.age<age>,
                        children.gender<sex>
                        using elementList, append;

            handler

                  case _EXCEPTION_DATABASE_SelectQueryFailed:
                        executionFailure = True;

                  case
              _EXCEPTION_DATABASE_InsufficientMemoryForQueryResult:
                        executionFailure = True;

                  case _EXCEPTION_DATABASE_FetchFailed:
                        executionFailure = True;

            endlayer;

            if (executionFailure)
                  executionFailure = False;
            output << "Select statement after insertion failed.\n";

            else
                  for (ITR = elementList; ITR; ITR++)
                        output << "Person : ";
                        output << ITR.Current().first << ' ';
                        output << ITR.Current().last << ' ';
                        output << ITR.Current().age << ' ';
                        output << ITR.Current().sex << '\n';
                  endfor;
            endif;

// ------------------------------------------------------------------ //
// Modify a row just inserted. Note that the container was
// populated above.
// ------------------------------------------------------------------ //

            layer<exceptionEventType>

                  for (ITR = elementList; ITR; ITR++)

                        if (ITR.Current().first == "Cheeta")

                          databaseUpdate<Duser, element : children>
                              children.name<first>,
                              children.last<last>

                              set name = "Jack", last = "Jackson"
                              where (name == {ITR.Current().first});

                          break;

                        endif;

                  endfor;
```

```
            handler

                  case _EXCEPTION_DATABASE_UpdateRequestFailed:
                        output << "Update of " <<
                          ITR.Current().first << " failed\n";

                        resume;

            endlayer;

// empty the container before select populates it

            -elementList;

            output << "After update ...\n\n";

            layer<exceptionEventType>

                  databaseSelect<Duser, element : children>
                        children.name<first>,
                        children.last<last>,
                        children.age<age>,
                        children.gender<sex>
                        using elementList, append;

            handler

                  case _EXCEPTION_DATABASE_SelectQueryFailed:
                        executionFailure = True;

                  case
                  _EXCEPTION_DATABASE_InsufficientMemoryForQueryResult:
                        executionFailure = True;

                  case _EXCEPTION_DATABASE_FetchFailed:
                        executionFailure = True;

            endlayer;

            if (executionFailure)
                  executionFailure = False;
                  output << "Select statement after update failed.\n";
            else
                  for (ITR = elementList; ITR; ITR++)
                        output << "Person : ";
                        output << ITR.Current().first << ' ';
                        output << ITR.Current().last << ' ';
                        output << ITR.Current().age << ' ';
                        output << ITR.Current().sex << '\n';
                  endfor;
            endif;

// ------------------------------------------------------------- //
// Delete rows just inserted. Note that the container was
// populated above. Also note that the first name "Cheeta"
// was changed to "Jack".
```

```
// ------------------------------------------------------------------ //

// Delete added rows, read and show

            layer<exceptionEventType>

                  for (ITR = elementList; ITR; ITR++)

                    if (ITR.Current().first == "Tarzan" ||
                        ITR.Current().first == "Jane" ||
                        ITR.Current().first == "Jack")

                          databaseRemove<Duser, element : children>
                              children.name<first>
                              where (name == {ITR.Current().first});

                    endif;

                  endfor;

            handler

                  case _EXCEPTION_DATABASE_RemoveRequestFailed:
                              output << "Removal of " <<
                               ITR.Current().first << " failed\n";
                              resume;

            endlayer;

// empty the container before select populates it

            -elementList;

            output << "After delete ...\n\n";

            layer<exceptionEventType>

                  databaseSelect<Duser, element : children>
                        children.name<first>,
                        children.last<last>,
                        children.age<age>,
                        children.gender<sex>
                        using elementList, append;

            handler

                  case _EXCEPTION_DATABASE_SelectQueryFailed:
                        executionFailure = True;

                  case
                  _EXCEPTION_DATABASE_InsufficientMemoryForQueryResult:
                        executionFailure = True;

                  case _EXCEPTION_DATABASE_FetchFailed:
                        executionFailure = True;

            endlayer;
```

```
            if (executionFailure)
                  executionFailure = False;
                  output << "Select statement after delete failed.\n";
            else
                  for (ITR = elementList; ITR; ITR++)
                        output << "Person : ";
                        output << ITR.Current().first << ' ';
                        output << ITR.Current().last << ' ';
                        output << ITR.Current().age << ' ';
                        output << ITR.Current().sex << '\n';
                  endfor;
            endif;

// This is the handler section for outermost exception layer

      handler

            case _EXCEPTION_DATABASE_UnsupportedDatabaseKind:

      output << "Caught _EXCEPTION_DATABASE_UnsupportedDatabaseKind\n";
                  executionFailure = True;

            case _EXCEPTION_DATABASE_LibraryInitializationFailed:
  output << "Caught _EXCEPTION_DATABASE_LibraryInitializationFailed\n";
                  executionFailure = True;

            case _EXCEPTION_DATABASE_ConnectionToServerFailed:
  output << "Caught _EXCEPTION_DATABASE_ConnectionToServerFailed\n";
                  executionFailure = True;

      endlayer;

      if (executionFailure)
            output << "Terminating program due to exception.\n";
      else output << "Program ended successfully.\n";
      endif;

      -elementList; // empty the container for cleanup

end;
```

The output of this example is as follows.

```
After insertion ...

Person : Borzoo Honargohar 24 M
Person : Cheeta Tree 3 M
Person : Jane Forest 23 F
Person : Payam Honargohar 36 M
Person : Peyvand Honargohar 34 M
Person : Syra Honargohar 4 F
Person : Tarzan Jungle 33 M
After update ...

Person : Borzoo Honargohar 24 M
```

```
Person : Jack Jackson 3 M
Person : Jane Forest 23 F
Person : Payam Honargohar 36 M
Person : Peyvand Honargohar 34 M
Person : Syra Honargohar 4 F
Person : Tarzan Jungle 33 M
After delete ...

Person : Borzoo Honargohar 24 M
Person : Payam Honargohar 36 M
Person : Peyvand Honargohar 34 M
Person : Syra Honargohar 4 F
Program ended successfully.
```

Appendix IV.9.D. Proxy Database Server

In Section IV.9.A.1. Connecting to Database, the library **struct** databaseType was introduced. The arguments passed to the constructor of **struct** databaseType initialize the following members.

```
int descriptor;
systemDatabaseKinds kind;
string host;
int port;
string database;
string connection;
int limit;
string proxy;
```

In the example of previous section we explained the first six parameters of the constructor, as seen in the following declaration.

```
databaseType Dbase(dbServerIP, dbPort,
      databaseName, vnode, _Database_Kind_Ingres);
```

The seventh parameter was explained earlier in Section IV.9..B.7. Fetch Statement, but will be illustrated in the example of this section. Here, we wish to explain the role of the eighth parameter, namely the member `proxy` of databaseType.

The default value for the eighth parameter is the empty string, meaning **no proxy**. In this model, the node that is executing the Z++ program communicates directly with the database server (locally or remotely).

In proxy model, the node that is executing the Z++ program communicates with the proxy server, which is responsible for communicating with the database server. The proxy server is just the **Z++ Internet Server**. That is, acting as a database proxy is simply one of the services of Z++ Internet Server. Therefore, the IP address for the **eighth** parameter is the IP address of a **Z++ Internet Server**.

For switching between the use of a proxy, or no proxy you simply need to provide an IP address for the eighth parameter, or leave it as the empty string (the default vale). This means that there is no need to make any other changes to the Z++ program.

Generally, the Z47 installed on small handheld devices requires the use of proxy database server. This is because the proxy server stores large amounts of data and provides it to the handheld device in small chunks, as will be illustrated in the example of this section. However, as far as database statements, the Z++ program **remains the same** for a large computer, and a computer with limited resources. The only difference in programs will be in their **presentation** phase, which depends on the capabilities of the device.

In the following example, the size of fetch is set to 2. There are only 4 rows in the table accessed. The select statement will only fetch 2 rows. After that, the fetch statement will fetch 2 rows at a time. When the size of fetch is set to a positive number, instead of default value of 0 (meaning fetch all), it is a good idea to call the **free** statement even if all rows are fetched. This was explained in Section IV.9.B.8. Free statement.

This example also provides an IP address for `ZppServerIP`, which is passed as the eighth argument to the constructor of `databaseType` as explained. The IP address is the address of a Z++ Internet Server. That is, the program was executed with a Z++ Internet Server running at the address, and serving as the proxy database server.

```
// Example.zpp

#include<Iterator.h>

#include<iostream.h>
using namespace ioSpace;

#include<database.h>
using namespace databaseSpace;

#include<exception.h>
using namespace exceptionSpace;

// ------------------------------------------------------------------ //

string dbServerIP = "192.168.1.64"; // IP address of Server
int dbPort = 21064;                 // port number
string databaseName = "demodb";     // Database name
string userName = "UserName";
string userPassWD = "PassWord";     // password if needed
string vnode = "Remote";            // name of virtual node

int FetchSize = 2;

// ZppServerIP is address of Proxy, a Z++ Internet Server.

string ZppServerIP = "192.168.1.64";

// ------------------------------------------------------------------ //
```

```
struct element
      string first;
      string last;
      short age;
      char sex;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void) throws(_EXCEPTION_EmptyContainer)

      using namespace ztlListIteratorSpace;
      List<element> elementList;

// The following constructor uses proxy and needs
// Z++ Internet Server up and running.

      databaseType Dbase(dbServerIP, dbPort, databaseName, "",
            _Database_Kind_Ingres, FetchSize, ZppServerIP);

// Database session begins here

      databaseUserType Duser(Dbase, userName, userPassWD);

      boolean MoreToFetch = True;

// Send a select query to the proxy

      databaseSelect<Duser, element : children>
            children.name<first>,
            children.last<last>,
            children.age<age>,
            children.gender<sex>
            using elementList, append;

      element et;

// Show the rows fetched by select, then continue to fetch
// the remaining rows from proxy and show them.

      do

            while (+elementList > 0)
                  et = elementList.Head();
                  output << "Person : ";
                  output << et.first << ' ';
                  output << et.last << ' ';
                  output << et.age << ' ';
                  output << et.sex << '\n';
            endwhile;

            if (!MoreToFetch)
                  output << "No more to fetch.\n";
                  break;
            else output << "\nFetching next " <<
```

```
                        FetchSize << " rows.\n\n";
            endif;

// Fetch statement returns false when there are no more rows to fetch

            MoreToFetch = databaseFetch<Duser, element : children>
            children.name<first>,
            children.last<last>,
            children.age<age>,
            children.gender<sex>
            using elementList, append;

      enddo;

// Since all rows are fetched there is nothing to free.
// However, after fetch the free statement is harmless
// even when not necessary.

      databaseFree<Duser>;

end;
```

The output of this example is as follows.

```
Person : Borzoo Honargohar 24 M
Person : Payam Honargohar 36 M

Fetching next 2 rows.

Person : Peyvand Honargohar 34 M
Person : Syra Honargohar 4 F
No more to fetch.
```

Appendix IV.9.E. Database kind specific parameters

Z++ database statements are **universal abstractions** for **uniform** interaction with all kinds of database servers. The only difference is in the initial values for constructors of library classes `databaseType` and `databaseUserType`.

The parameters for `databaseType` that depend on the kind of database are those that initialize the following members.

```
systemDatabaseKinds kind;
string host;
int port;
string connection;
```

The parameters for the constructor of `databaseUserType` are user-name and password, which do not need explanation. In the following subsections commonly known values for the four parameters mentioned here will be discussed. For instance, port number depends on your particular set up.

Appendix IV.9.E.1. MySQL Parameters

Values of parameters for MySQL are as follows.

```
kind : _Database_Kind_Mysql
host : IP address of database server
port : 3306
connection : empty string
```

Appendix IV.9.E.2. Ingres Parameters

Values of parameters for Ingres are as follows.

```
kind : _Database_Kind_Ingres
host : IP address of database server
port : 21064
connection : vnode
```

For Ingres, the value for the member `connection` is the name of virtual node (vnode) as discussed in Ingres manuals.

Appendix IV.9.E.3. Postgres Parameters

Values of parameters for Postgres are as follows.

```
kind : _Database_Kind_Postgres
host : IP address of database server (usually 127.0.0.1)
port : 5432
connection : empty string
```

Note that Postgres server uses IP address “127.0.0.1”. When your Z++ program runs on the same node as the server do not use the IP address of the node.

When using Z++ Internet Server as proxy, and the Internet Server runs on the same node as the Postgres database server, use the string “127.0.0.1” for the host. However, the IP address for proxy will be the IP address of the node on which Z++ Internet Server is running.

Appendix IV.9.F. Database Exceptions

System header file exception.h defines all exceptions. Below is the list of exceptions specific to database statements.

```
_EXCEPTION_DATABASE_UnsupportedDatabaseKind
_EXCEPTION_DATABASE_LibraryInitializationFailed
_EXCEPTION_DATABASE_ConnectionToServerFailed
_EXCEPTION_DATABASE_SelectQueryFailed
_EXCEPTION_DATABASE_InsertRequestFailed
_EXCEPTION_DATABASE_RemoveRequestFailed
_EXCEPTION_DATABASE_UpdateRequestFailed
_EXCEPTION_DATABASE_InsufficientMemoryForQueryResult
_EXCEPTION_DATABASE_FetchFailed
```

Appendix IV.9.G. Datatypes for columns

All Z++ numeric types and **string** type are supported. Database manuals refer to **string** type as **varchar**. More specifically, numeric types are **char**, **uchar**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float** and **double**.

Appendix V. Z++ Template Library (ZTL)

The implementation of ZTL is an example for using several linguistic abstractions, in particular the **namespace** for packaging libraries.

The packaging of ZTL library begins with file ztlException.h that includes the following.

```
namespace ztlExceptionSpace

enum ztlExceptionType : exceptionSpace::exceptionEventType {
      _EXCEPTION_EmptyContainer,
      _EXCEPTION_IncompatibleVectors,
      _EXCEPTION_VectorIndexOutOfBound,
      _EXCEPTION_HashItemNotFound,
      _EXCEPTION_HashItemIndexOutOfBound,
      _EXCEPTION_LastNotUsedContainer
};

endspace;
```

As usual, this defines the exceptions used in ZTL by **extending** system exceptions `exceptionEventType`. The name of ZTL exceptions is ztlExceptionType.

The file ztlException.h also introduces the **namespace** `ztlExceptionSpace` from which all ZTL namespaces are **derived**.

The following sections present ZTL. Remember that a **protected namespace** cannot be used directly, and is only intended for derivation by another **namespace**.

Appendix V.A. List, Stack and Queue

Include file is: **Iterator.h**.
Namespace is: **ztlListIteratorSpace**.

The **namespace** ztlListIteratorSpace contains List, Stack and Queue, and not just List **template**. The Stack and Queue templates use the List **template**.

Before defining each **template class**, the file Iterator.h begins with a couple of templates used in the construction of the List template, shown below.

```
protected namespace ztlBaseContainerSpace : ztlExceptionSpace

template<type T> struct Template_Element
      T Item;
      Template_Element<T>* link;

      Template_Element(void);
      Template_Element(const T&);
```

```
end;

template<type T> struct Container
      uint Size;  //size of container

      Container(uint);
      uint operator+(void) const;       //size
      void operator-(void);             //make empty
      boolean operator%(const T&) const;    //membership
end;

template<type T> class Iterator end;  //forward declaration

endspace;
```

The **namespace** ztlBaseContainerSpace is specified **protected** and is meant for internal use.

The **struct** Template_Element will always keep a **copy** of the item in any container, List, Stack or Queue. The **struct** Container provides the basic operations for all containers. The membership **operator** % is for finding out whether an item belongs to a container or not.

The **class** Iterator is a forward declaration, without members and methods.

Appendix V.A.1. List

The definition of the List **template class** is preceded with the definition of **template** BaseList for its internal use. The **struct** BaseList also introduces the **friend** Iterator **class**. The **template class** Iterator traverses List.

```
template<type T> struct BaseList : public Container<T>
      ~BaseList(void);

protected:
      friend Iterator<T>;
      Template_Element<T>* last;

//Methods

      BaseList(const BaseList&);
      void Cleanup(void);
      void copy(const BaseList&);
      void Clear(void);
      void insert(Template_Element<T>*);
      void append(Template_Element<T>*);
      T head(void);
      T tail(void);
      void remove(const T&);
      void replace(const T&);
      boolean member(const T&) const;
```

```
      void operator+=(const BaseList<T>&);
      boolean operator==(const BaseList<T>&) const;
      boolean operator!=(const BaseList<T>&) const;
      const BaseList<T>& Self(void) const;
end;
```

This is the List template for use in programs.

The **template class** Iterator is discussed later, in Appendix Iterator.

```
template<type T> class List : public BaseList<T>
      operator const BaseList<T>&(void) const;

public:

      ~List(void);
      uint operator+(void) const;
      List<T>& operator-(void);
      List<T>& operator=(const List<T>&);
      boolean operator%(const T&) const;
      List<T>& operator<<(const T&);
      List<T>& insert(const T&);
      List<T>& operator>>(const T&);
      List<T>& operator-(const T&);
      List<T>& operator+(const T&);
      List<T>& append(const T&);
      List<T>& makeHead(const T&);
      const T& top(void) throws(_EXCEPTION_EmptyContainer);
      T& First(void) throws(_EXCEPTION_EmptyContainer);
      const T& bottom(void) throws(_EXCEPTION_EmptyContainer);
      T& Last(void) throws(_EXCEPTION_EmptyContainer);
      T Head(void) throws(_EXCEPTION_EmptyContainer);
      T Tail(void) throws(_EXCEPTION_EmptyContainer);
      List<T>& operator+=(const List<T>&);
      boolean operator==(const List<T>&) const;
      boolean operator!=(const List<T>&) const;
end;
```

Description of Methods

In order to describe methods of the List **template class**, let L be an instantiation of the List **template**, and let e be an element (item) of the type instantiating the parameter T of the List.

`L << e`. Inserts e at head of L.
`insert(e)`. Same as `operator<<` above.

`L + e`. Appends e to tail of L.
`append(e)`. Same as `operator+` above.

`L - e`. Removes the first occurrence of e, if any, from L.

`L >> e`. Removes all occurrences of e from L.

`L += S`. Adds the List S to the tail of List L, without changing List S.

`makeHead(e)`. If e is not in L, inserts e at head of L. Otherwise, the first occurrence of e in L is made the head of L.

`Head()`. Returns the first element in L, and removes it from the List L.
`Tail()`. Returns the last element in L, and removes it from the List L.

`top()`. Returns a **const** reference to the first element in L.
`bottom()`. Returns a **const** reference to the last element in L.
`First()`. Returns a reference to the first element in the List L.
`Last()`. Returns a reference to the last element in the List L.

`+L`. Returns the number of items in the List L.
`-L`. Makes the List L empty.
`L % e`. Membership. Returns 1 if e is in the List L, else returns 0.
`L = S`. Assignment. Assigns the List S to the List L, making L same as S.
`L == S`. Equality. Returns true if Lists L and S contain the same items, in same order.
`L != S`. Inequality. The negation of `operator==`, above.

Appendix V.A.2. Stack

The **template class** Stack uses List **template**. The **template class** Iterator is discussed later, in Appendix Iterator.

```
template<type T> class Stack
protected:
      friend Iterator<T>;
      List<T> Elms;
public:
      Stack(void);
      Stack(const List<T>&);
      Stack(const Stack<T>&);
      ~Stack();

      Stack<T>& operator<<(const T&);
      Stack<T>& push(const T&);
      T pop(void) throws(_EXCEPTION_EmptyContainer);
      const T& top(void) throws(_EXCEPTION_EmptyContainer);

      Stack<T>& operator-(void);
      uint operator+(void) const;
      boolean operator%(const T&) const;
      Stack<T>& operator+=(const Stack<T>&);
      Stack<T>& operator=(const Stack<T>&);
      boolean operator==(const Stack<T>&) const;
      boolean operator!=(const Stack<T>&) const;
```

```
end;
```

Description of Methods

Let S be an instantiation of the Stack **template**, and let e be an element (item) of the type instantiating the parameter T of the Stack.

A constructor allows S to construct itself from a List.

`S << e`. Pushes e on to Stack S.
`push(e)`. Same as `operator<<` above.

`pop()`. Pops the Stack S and returns the element popped.

`top()`. Returns a **const** reference to the top of the Stack S.

`S += R`. Attaches the Stack R to end of Stack S, without changing R.

`+S`. Returns the number of items in the Stack S.
`-S`. Makes the Stack S empty.
`S % e`. Membership. Returns 1 if e is in the Stack S, else returns 0.

`S = R`. Assignment. Assigns the Stack R to the Stack S, making S same as R.
`S == R`. Equality. Returns true if Stacks S and R contain the same items, in same order.
`S != R`. Ineqaulity.The negation of `operator==`, above.

Appendix V.A.3. Queue

The **template class** Queue uses List **template**. The **template class** Iterator is discussed later, in Appendix Iterator.

```
template<type T> class Queue
protected:
      friend Iterator<T>;
      List<T> Elms;
public:
      Queue(void);
      Queue(const List<T>&);
      Queue(const Queue<T>&);
      ~Queue();

      Queue<T>& operator<<(const T&);
      Queue<T>& push(const T&);
      T pop(void) throws(_EXCEPTION_EmptyContainer);
      const T& top(void) throws(_EXCEPTION_EmptyContainer);

      Queue<T>& operator-(void);
      uint operator+(void) const;
      boolean operator%(const T&) const;
```

```
        Queue<T>& operator+=(const Queue<T>&);
        Queue<T>& operator=(const Queue<T>&);
        boolean operator==(const Queue<T>&) const;
        boolean operator!=(const Queue<T>&) const;
end;
```

Description of Methods

Let Q be an instantiation of the Queue **template**, and let e be an element (item) of the type instantiating the parameter T of the Queue.

A constructor allows Q to construct itself from a List.

`Q << e`. Pushes e on to Queue Q.
`push(e)`. Same as `operator<<` above.

`pop()`. Pops the Queue Q and returns the element popped.

`top()`. Returns a **const** reference to the top of the Queue Q.

`Q += R`. Attaches the Queue R to end of Queue Q, without changing R.

`+Q`. Returns the number of items in the Queue Q.
`-Q`. Makes the Queue Q empty.
`Q % e`. Membership. Returns 1 if e is in the Queue Q, else returns 0.

`Q = R`. Assignment. Assigns the Queue R to the Queue Q, making Q same as R.
`Q == R`. Equality. Returns true if Queues Q and R contain the same items, in same order.
`Q != R`. Ineqaulity.The negation of `operator==`, above.

Appendix V.A.4. Iterator

The **template class** Iterator can be used with **template class** List, Stack and Queue. That is, you can declare the Iterator for use with Stack, and later use the same Iterator for traversing List or Queue, so long as the **instantiators are the same**.

The **template class** Iterator uses the following enumeration to set, or reset itself to the kind of container it is iterating.

```
enum containerTargetKind {_List, _Queue, _Stack};
```

This is the definition of the **template class** Iterator.

```
template<type T> class Iterator

      Iterator<T>& operator=(const Iterator<T>&);

protected:

      Iterator(const Iterator<T>&);
      containerTargetKind kind;
      void* Root; //points to Container
      Template_Element<T>* current;

public:

      Iterator(void);
      Iterator(const List<T>&);
      Iterator(List<T>*);
      Iterator(const Queue<T>&);
      Iterator(Queue<T>*);
      Iterator(const Stack<T>&);
      Iterator(Stack<T>*);
      Iterator<T>& Reset(void);

      Iterator<T>& operator=(const List<T>&);
      Iterator<T>& operator=(List<T>*);
      Iterator<T>& operator=(const Queue<T>&);
      Iterator<T>& operator=(Queue<T>*);
      Iterator<T>& operator=(const Stack<T>&);
      Iterator<T>& operator=(Stack<T>*);

      const T& Current(void);
      T& Object(void);
      boolean Last(void);
      operator void*(void);
      Iterator<T>& operator+@(void); //postfix ++ operator
      Iterator<T>& operator++(void);
end;
```

Description of Methods of Iterator

A container C in this context will refer to List, Stack or Queue. Let I be an Iterator.

You can declare an Iterator I using one of its constructors that takes a container C, or a pointer to a container C. You can also declare an Iterator I using its default constructor.

However you construct the iterator I, it is safer to use the following assignments prior to start of iteration.

`I = C`. Resets I for iterating container C.
`I = P`. Same as above, when P is a pointer to a container C.

Once the Iterator I is set to iterate a container C, the following methods are available.

I++ and ++I. Move I to the next element in C.

I. This is a **boolean**. It return true so long as I++ can move forward. When it returns false all elements in the container C have been visited.

`Last()`. Returns true when exactly one more I++ can be done before visiting all elements in C. That is, when iteration has reached the element just before the last. That means after one more I++, the test I will return false.

`Reset()`. Resets I so iteration begins from the start of container C.

`Current()`.Returns a **const** reference to the element in C that I is currently pointing to.
`Object()`. Returns a reference to the element in C that I is currently pointing to.

Appendix V.A.5. Example

The following example uses a single iterator for traversing a List, a Stack and a Queue. It also shows the use of some of the methods of these templates.

```
// Example.zpp

#include<Iterator.h>
#include<iostream.h>
using namespace ioSpace;

// Remark. Iterator.h includes List, Stack and Queue.

// ------------------------------------------------------------------ //
// Let us define a structure that we are going to put into containers.

struct element
      int n;
      double b;

      element(void);
      element(const element&);
      element(int, double);
      friend outStream& operator<<(outStream&, const element&);
end;

element::element(void)
      n = 47;
      b = 3.5;
end;

element::element(const element& e)
      n = e.n;
      b = e.b;
end;

element::element(int i, double d)
      n = i;
      b = d;
end;

outStream& operator<<(outStream& o, const element& e)
      o << "Member n is : " << e.n <<'\n';
      o << "Member b is : " << e.b <<'\n';
      return o;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////
// Operator + in front of any container returns its size.

entry void main(void) throws(_EXCEPTION_EmptyContainer)
```

```
      output << "Hello World!\n";

// Open List, Stack and Queue namespaces

      using namespace ztlListIteratorSpace;

// Instantiate List template with type element

      List<element> elementList;

// List is just created, so it is empty. Let us cause an exception

      layer<ztlExceptionType>
            elementList.Head();
      handler
            case _EXCEPTION_EmptyContainer:
                  output << "Caught _EXCEPTION_EmptyContainer.\n";
      endlayer;

// Create two objects D, N and insert them in List in that order

      element D;
      element N(3, 7.9);
      elementList << D << N;

// Since D was inserted first, it is last element in list.
// Next line will output the tail of list, which is D.

      output << elementList.bottom();

// Overwrite members of D by top of List, i.e. N
// and output the result.

      D = elementList.First();
      output << D;

// List has its own copy, so D in List did not change

      output << elementList.bottom();

      elementList << element(17, 95.3); //insert an object on the fly

      Iterator<element> I; //create an iterator

      output << "\nPrinting all list elements via iteration.\n";

      for (I = elementList; I; I++) output << I.Current(); endfor;

      output <<
"\nPrinting all list elements while removing them from list.\n";

      while (+elementList > 0) output << elementList.Head(); endwhile;

// ------------------------------------------------------------- //

      output << "\nStarting Queue.......\n";
```

```
      Queue<element> elementQueue;
      elementQueue << D << N << element(17, 95.3); //push, here D == N

      output << "\nPrinting all queue elements via iteration.\n";

      for (I = elementQueue; I; I++) output << I.Current(); endfor;

      output <<
"\nPrinting all queue elements while removing them from queue.\n";

      while (+elementQueue > 0) output << elementQueue.pop(); endwhile;

// ------------------------------------------------------------------ //

      output << "\nStarting Stack.......\n";

      Stack<element> elementStack;
      elementStack << D << N << element(17, 95.3); //push, here D == N

      output << "\nPrinting all stack elements via iteration.\n";

      for (I = elementStack; I; I++) output << I.Current(); endfor;

      output <<
"\nPrinting all stack elements while removing them from stack.\n";

      while (+elementStack > 0) output << elementStack.pop(); endwhile;

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Caught _EXCEPTION_EmptyContainer.
Member n is : 47
Member b is : 3.5
Member n is : 3
Member b is : 7.9
Member n is : 47
Member b is : 3.5

Printing all list elements via iteration.
Member n is : 17
Member b is : 95.3
Member n is : 3
Member b is : 7.9
Member n is : 47
Member b is : 3.5

Printing all list elements while removing them from list.
Member n is : 17
Member b is : 95.3
Member n is : 3
Member b is : 7.9
```

```
Member n is : 47
Member b is : 3.5

Starting Queue.......

Printing all queue elements via iteration.
Member n is : 3
Member b is : 7.9
Member n is : 3
Member b is : 7.9
Member n is : 17
Member b is : 95.3

Printing all queue elements while removing them from queue.
Member n is : 3
Member b is : 7.9
Member n is : 3
Member b is : 7.9
Member n is : 17
Member b is : 95.3

Starting Stack.......

Printing all stack elements via iteration.
Member n is : 17
Member b is : 95.3
Member n is : 3
Member b is : 7.9
Member n is : 3
Member b is : 7.9

Printing all stack elements while removing them from stack.
Member n is : 17
Member b is : 95.3
Member n is : 3
Member b is : 7.9
Member n is : 3
Member b is : 7.9
Good-bye World!
```

Appendix V.B. Vector

Include file is: **Vector.h**.
Namespace is: **ztlVectorSpace**.

Z++ already supports two forms of dynamic arrays discussed in Dynamic Arrays and New Operator. However, once such an array is created it is not possible to change its size. The **template class** Vector is an array, which provides a way for expanding and shrinking its size.

The following definitions precede the definition of Vector **template**.

```
template<type T> typedef degenrateTemplate T[];

template<type T> struct Vector_Element
      degenrateTemplate<T>* array;
      Vector_Element<T>* link;

      Vector_Element(int);
      ~Vector_Element(void);
end;
```

The **struct** `Vector_Element` is a linked list, and its member `array` is a pointer to an array. The **class Vector** uses this linked list to expand and shrink itself. Explanations follow the definition.

```
template<type T> class Vector

protected:

      Vector_Element<T>* last;
      int Initial;
      int Increment;
      int Size;

      const Vector<T>& Self(void) const;
      boolean structure(const Vector<T>&);
      void clear(void);

public:

      Vector(int i = 32, int j = 32);
      Vector(const Vector<T>&);
      ~Vector(void);

      void operator~(void);
      int operator+(void) const;

      T& operator[](int i) {
            (i > 0 || i <= Size) _EXCEPTION_VectorIndexOutOfBound;
      };
```

```
        const T& operator[](long i) const {
              (i > 0 || i <= Size) _EXCEPTION_VectorIndexOutOfBound;
        };

        int operator<<(int i) {
              (i > 0) _EXCEPTION_VectorIndexOutOfBound;
        };

        int operator>>(int i) {
              (i > 0 || i <= Size) _EXCEPTION_VectorIndexOutOfBound;
        };

        void operator=(const T&);
        Vector<T>& operator=(const Vector<T>&);
        boolean operator==(const T&);

        boolean operator==(const Vector<T>& a) const {
              (structure(a)) _EXCEPTION_IncompatibleVectors;
        };

        boolean operator!=(const T&);

        boolean operator!=(const Vector<T>& a) const {
              (structure(a)) _EXCEPTION_IncompatibleVectors;
        };

end;
```

The members Initial and Increment receive the default value of 32, unless you override them. Initial is the initial number of cells, and Increment is the number of cells to add, or remove when expanding or shrinking (an instance of) Vector.

While Vector can change its size **dynamically**, this needs to be done **manually**. The member Size is the total number of cells of Vector at any time.

Suppose you are about to set the value of some cell, of Vector Vec, at index n. You are not sure Vec has n cells. Before assigning to the cell you can use the expansion method, as shown below.

```
Vec << n;
Vec[n] = Something;
```

If n is less than Size, no expansion will take place. Otherwise, assuming the default values of 32, suppose Size is 32 and n is 65. This will add two increments making the Size of Vec 96. That is, Vec will be expanded by increments of 32 until its Size exceeds the index n.

The shrink method removes increments until the indicated index is in an increment so that it is less than the member Size. For instance, for our Vec with Size 96, the following will remove one increment making Size equal to 64.

```
Vec >> 40;
```

The shrink method will not remove the initial number of cells, and if its argument is less than Size, nothing will happen.

Three methods, expansion (`operator<<`), shrinking (`operator>>`) and `operator[]` are constrained. If constraints are violated exception `_EXCEPTION_VectorIndexOutOfBound` will be raised.

The method `operator[]` has two overloaded versions. The one, which takes a **long** argument instead of an **int** argument, is a **constant** method that returns a **constant** reference. When you need to call one or the other version use a constructor **cast**, like **int**(0) or **long**(3). For instance, the implementation of equality **operator** for Hash Table uses **long**, while the implementation of Heap methods Head() and Tail(), use **int**.

The **protected** method structure() checks for compatibility of array in assignment and comparison operations. It is used as constraint for relevant methods, and its violation will raise the exception `_EXCEPTION_IncompatibleVectors`.

The **protected** method clear() is used internally, for instance by destructor. For putting an instance of Vector back to its **initial state**, use the **public** method `operator~()`. Note that this will **destroy all** objects that are values of cells of Vector, putting it back to the time that it was declared, with **no** cell having a **valid** object.

Method `operator+()` returns the value of member Size. This is the total number of cells currently in Vector.

An overloaded version of **operator** assign allows assigning a **single** object to **all** cells of Vector. This is useful for initialization of cells.

In the following example note that the include file for Vector comes **before** the **using** statement. This is because ZTL does not separate the implementation of templates from their definition. That is, the implementation is in the same file as the definition. This is only for keeping things simple for initial users. However, the compiler does not allow the implementation of a **namespace** when any **namespace** is opened via the **using** statement. Remember that you can always **close** a **namespace** via **endusing**, and reopen it if necessary.

```
// Sample.zpp

#include<iostream.h>
#include<Vector.h>

using namespace ioSpace;

// ------------------------------------------------------------------ //
// A structure for array instantiator.
```

```
struct element
	int n;
	double b;

	element(void);
	element(const element&);
	element(int, double);
	friend outStream& operator<<(outStream&, const element&);
end;

element::element(void)
	n = 47;
	b = 3.5;
end;

element::element(const element& e)
	n = e.n;
	b = e.b;
end;

element::element(int i, double d)
	n = i;
	b = d;
end;

outStream& operator<<(outStream& o, const element& e)
	o << "Member n is : " << e.n <<'\n';
	o << "Member b is : " << e.b <<'\n';
	return o;
end;

////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

	output << "Hello World!\n";

	using namespace ztlVectorSpace;

	Vector<element> elemVector;

	elemVector << 65; // expand
	output << +elemVector << '\n'; // 96

	elemVector >> 40; // shrink
	output << +elemVector << '\n'; // 64

	elemVector = element(47, 97.47); // set all cells

	elemVector[60] = element(77, 77.77); // change a cell

	output << "Values of cell 32 are:\n" << elemVector[32] << '\n';
	output << "Values of cell 60 are:\n" << elemVector[60] << '\n';

// Next line is not necessary. Destructor will do it.
```

```
        ~elemVector; // clear, remove all element

        endusing namespace ztlVectorSpace;

        output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
96
64
Values of cell 32 are:
Member n is : 47
Member b is : 97.47

Values of cell 60 are:
Member n is : 77
Member b is : 77.77

Good-bye World!
```

Appendix V.C. Array

Include file is: **Array.h**.
Namespace is: **ztlArraySpace**.

The **template class** Array is a specialization of **template class** Vector to discrete numeric types. This specialization is enforced by the following **template pattern**.

```
pattern template<T> ztlArrayPattern
      T : char, uchar, short, ushort, int, uint, long, ulong;
end;
```

Below is the definition of **template class** Array. Explanations will follow the definition.

```
template<type T : ztlArrayPattern> class Array : private Vector<T>

      operator const Vector<T>&(void) const;

public:

      Array(int i = 32, int j = 32);
      Array(const Array<T>&) throws(_EXCEPTION_VectorIndexOutOfBound);
      ~Array(void);

      void operator~(void);
      int operator+(void) const;

      T& operator[](int) throws(_EXCEPTION_VectorIndexOutOfBound);
      const T& operator[](long) const
            throws(_EXCEPTION_VectorIndexOutOfBound);

      int operator<<(const int)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      int operator>>(const int)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      void operator=(const T&); // pass is by reference

      Array<T>& operator=(const Array<T>&)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      Array<T>& operator+=(const Array<T>& a) {
            (structure(a)) _EXCEPTION_IncompatibleVectors;
      };
      Array<T>& operator-=(const Array<T>& a) {
            (structure(a)) _EXCEPTION_IncompatibleVectors;
      };
      Array<T>& operator*=(const Array<T>& a) {
            (structure(a)) _EXCEPTION_IncompatibleVectors;
      };
      Array<T>& operator/=(const Array<T>& a) {
            (structure(a)) _EXCEPTION_IncompatibleVectors;
```

```
        };

        Array<T>& operator+=(T);
        Array<T>& operator-=(T);
        Array<T>& operator*=(T);
        Array<T>& operator/=(T);
        Array<T>& operator-(void);

        Array<T>& operator&=(const Array<T>& a) {
              (structure(a)) _EXCEPTION_IncompatibleVectors;
        };
        Array<T>& operator|=(const Array<T>& a) {
              (structure(a)) _EXCEPTION_IncompatibleVectors;
        };
        Array<T>& operator^=(const Array<T>& a) {
              (structure(a)) _EXCEPTION_IncompatibleVectors;
        };
        Array<T>& operator<<=(const Array<T>& a) {
              (structure(a)) _EXCEPTION_IncompatibleVectors;
        };
        Array<T>& operator>>=(const Array<T>& a) {
              (structure(a)) _EXCEPTION_IncompatibleVectors;
        };

        Array<T>& operator&=(T);
        Array<T>& operator|=(T);
        Array<T>& operator^=(T);
        Array<T>& operator<<=(T);
        Array<T>& operator>>=(T);
        Array<T>& operator++(void);
        Array<T>& operator+@(void); // postscript ++ overloaded
        Array<T>& operator--(void);
        Array<T>& operator-@(void); // postscript -- overloaded

        boolean operator==(const T&);

        boolean operator==(const Array<T>&)
              throws(_EXCEPTION_IncompatibleVectors);

        boolean operator!=(const T&);

        boolean operator!=(const Array<T>&)
              throws(_EXCEPTION_IncompatibleVectors);
end;
```

The **template class** Array overloads all Assignment Operators for arguments of type T, as well as Arrays of T (the **template** parameter). It also overloads the comparison operators for **equality** and **inequality** the same way.

Because Array inherits from Vector, the following assignment uses pass by reference.

```
void operator=(const T&); // pass is by reference
```

In case of Vector this is necessary because T could be instantiated with large types. However, Array instantiators are only numeric types. The only problem that this causes is that literals cannot be used. This is illustrated in following example.

```
//Array.zpp

#include<iostream.h>
#include<Array.h>

using namespace ioSpace;
using namespace ztlArraySpace;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      int init = 47;
      Array<int> IntArray;
      IntArray = init; // need object, not literal value
      IntArray += 50; // here, literal or object are fine

      output << IntArray[20] << '\n';

      init = 7;
      Array<long> LongArray;
      LongArray = init;

// Unlike arrays of built-in type, we cannot do
//     LongArray *= IntArray;
// Implicit conversions only work on regular arrays.

      LongArray *= LongArray;

      output << LongArray[10] << '\n';

      LongArray++;
      LongArray <<= 1;

      output << LongArray[15] << '\n';

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
97
49
100
Good-bye World!
```

Appendix V.D. Hash Table

Include file is: **Hash.h**.
Namespace is: **ztlHashSpace**.

The abstract data type **Hash Table** is quite useful. A compiler's symbol table is a hash table.

The structure of Hash in ZTL uses a Vector of Lists. The default size of the vector is 97, a prime number. The size of the Vector does not change. Instead, in case of collisions the List at the cell stores new objects. The List acts like a **stack** in that the last object stored in the List at a particular cell is the first to visit. However, all objects in the List can be reached.

The **template class** Hash comes with an Iterator. The order of visiting elements in the table is not predetermined. Nonetheless, the Iterator will visit **all** the elements before indicating that it has reached the end of iteration.

Objects that can instantiate the **template class** Hash must satisfy the conditions of the following **template pattern**.

```
pattern template<T> ztlHashPattern
      T(string);
      operator const string&(void) const;
end;
```

The **pattern** requires that the **instantiator** have a constructor taking a **string** object, and to provide a conversion **operator** to **string** type, as specified.

ZTL provides the following default Hash Function, which can be over-ridden. However, the function must take a **string**, and return an **int**.

```
type int zHashFun(const string&); // Type of hash-function

int zDefaultHasher(const string& s) // Default hash-function
      int i = 0;
      int j = size(s);
      while (--j >= 0) i = i << 1 ^ getchr(s, j); endwhile;
      return i < 0 ? -i : i;
end;
```

Below is the definition of **template class** Hash. The definition is preceded with a forward declaration for Iterator. Explanations follow the definition.

```
template<type T> class HashIterator end; // Forward declaration
```

```
template<type T : ztlHashPattern> class Hash

protected:

      friend HashIterator<T>;
      Vector<List<T>* > H;
      zHashFun F;
      int Hsize;
      Iterator<T> I;

public:

      Hash(int n = 97, zHashFun f = 0);
      Hash(const Hash<T>&);
      ~Hash();
      Hash<T>& operator-(void);
      uint operator+(void);
      Hash<T>& operator<<(const T&);
      Hash<T>& operator>>(const T&)throws(_EXCEPTION_HashItemNotFound);
      Hash<T>& operator>>(string) throws(_EXCEPTION_HashItemNotFound);
      const T& Current(void);
      T& Object(void);
      boolean operator%(const T&);
      boolean operator%(string);
      int operator%=(const T&);
      int operator%=(string);
      T& operator/(const T&) throws(_EXCEPTION_HashItemNotFound);
      T& operator/(string) throws(_EXCEPTION_HashItemNotFound);
      Hash<T>& operator=(const Hash<T>&);
      boolean operator==(const Hash<T>&) const;
      boolean operator!=(const Hash<T>&) const;

      T& Member(const T&, int)
            throws(_EXCEPTION_HashItemIndexOutOfBound,
                  _EXCEPTION_HashItemNotFound);

      T& Member(string, int)
            throws(_EXCEPTION_HashItemIndexOutOfBound,
                  _EXCEPTION_HashItemNotFound);
end;
```

For purposes of explanation of methods, let H be a Hash Table and let e be an element of type T (**template parameter**). In the following, due to overloading, whenever e appears as argument, an object of type **string** can also be passed instead of e.

-H makes the Hash Table H empty: `operator-(void)`.
+H returns number of elements in H: `operator+(void)`.
H % e returns true only when e is an item in H.
H << e inserts e in H.
H >> e removes first (top, most recent) occurrence of e from H.

H / e returns a **none-constant** reference to e, into the table. Normally, you would pass a **string** identifying e in order to get a reference to the item e.

Only, **immediately** after the call H % e returns **true**, the following two methods are valid. The call H % e makes preparations for theses methods by setting its **current** element to the argument e. This is true even when you pass a **string** to the call. If the call returns **true** the **current** element will be set to the item e identified by the **string** you passed to the call.

Current() returns a **constant** reference to **current** element.
Object() returns a **none-constant** reference to **current** element

H %= e returns number of occurrences of e in the table. Instead of e, a string identifying e will do the same. The number returned by this **operator** can be used to reach **all occurrences** of e in the Hash Table H. The following method does that, where n is a **positive integer** 1,2, 3, and so on.

Member(e, n) This method returns a **none-constant** reference to the n-th occurrence of e into the table. Note that e can be replaced with a **string**.

```
// Example.zpp

#include<Hash.h>
using namespace ztlHashSpace;

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //
// HashElement is type of objects that will go into Hash Table.

class HashElement
      string name<visible>;
      int age<visible>;

public:

      HashElement(string);
      HashElement(string, int);
      operator const string&(void) const;
end;

HashElement::HashElement(string s)
      name = s;
      age = 0;
end;

HashElement::HashElement(string s, int n)
      name = s;
      age = n;
end;

HashElement::operator const string&(void)
      return name;
end;
```

```
////////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
////////////////////////////////////////////////////////////////////////

entry void main(void)

  Hash<HashElement> HashTable;

  HashElement e_1("Jack", 58);
  HashElement e_2("Jeronimo", 17);

  HashTable << e_1;
  HashTable << e_2;

  if (HashTable % "Jeronimo")
      output << "Jeronimo is in the table.\n";
      output << "He is " << HashTable.Object().age << " years old.\n";
  else output << "Jeronimo Not Found in the table.\n";
  endif;

  for (HashIterator<HashElement> I = HashTable; I; I++)
      output << I.Object().name << " is " <<
                I.Object().age << " years old.\n";
  endfor;


  output << "Size of table is : " << +HashTable << '\n';
  -HashTable;
  output << "Size of table after clean up is : " << +HashTable << '\n';

end;
```

The output of this example is as follows.

```
Jeronimo is in the table.
He is 17 years old.
Jeronimo is 17 years old.
Jack is 58 years old.
Size of table is : 2
Size of table after clean up is : 0
```

Appendix V.E. Heap

Include file is: **Heap.h**.
Namespace is: **ztlHeapContainerSpace**.

The **template class Heap** is a **priority queue**, as well as a **priority stack**. Either as a queue, or a stack, the next element to be retrieved depends on its priority. The Heap **template class** uses the following **template** as a representation of its items.

```
template<type T, type P> struct Heap_Element
      P priority;
      List<T>* list;

// Methods

      Heap_Element(P);
      ~Heap_Element(void);

      void operator-(void);
      void operator~(void);
      boolean operator%(const T&);

      Heap_Element<T,P>& operator=(const Heap_Element<T,P>&);
      void operator<<(const T&);
      void operator+(const T&);
      void operator>>(const T&);

      const T& top(void) throws(_EXCEPTION_EmptyContainer);
      T Head(void) throws(_EXCEPTION_EmptyContainer);
      T Tail(void) throws(_EXCEPTION_EmptyContainer);

      int operator+(void) const;
      boolean operator==(const Heap_Element<T,P>&) const;
end;
```

The type parameter T is the type of item to be stored in the Heap, and the type parameter P is the priority of this item. The parameter P requires an argument that possesses a comparison operator, like numeric types.

The **template struct** Heap_Element is a List of items with same priority. The List can be used as a queue or as a stack. For a **queue** action use **operator+** to **append** to the tail of the List, and for a **stack** action, use **operator<<** to insert at the head of the List. **Operator>>** always removes the item at the head of the List. However, this structure is used by, the **template class** Heap, which also provides similar methods.

The **template class** Heap uses a Vector of Heap_Element. The size of Vector is dynamically adjusted, based on number of priorities. That is, the size of Vector depends on the number of priorities, not the number of items in the Heap. All items with same priority are stored in the List of Heap_Element for that priority.

Below is the definition of **template class** Heap, and explanations will follow.

Note that priority P is of **discrete** numeric type. The **lower** the numeric value, the **higher** the priority.

```
template<type T, type P : ztlHeapPattern> class Heap

protected:

      int Increment;
      int Size;
      Vector<Heap_Element<T,P> > H;

// Private Methods

      void resize(void) throws(_EXCEPTION_VectorIndexOutOfBound);
      int percolate(P) throws(_EXCEPTION_VectorIndexOutOfBound);
      void move(int) throws(_EXCEPTION_VectorIndexOutOfBound);
      int sift(P) throws(_EXCEPTION_VectorIndexOutOfBound);
      int down(int) throws(_EXCEPTION_VectorIndexOutOfBound);
      void adjust(void) throws(_EXCEPTION_VectorIndexOutOfBound);
      int find(P) throws(_EXCEPTION_VectorIndexOutOfBound);
      void remove(int) throws(_EXCEPTION_VectorIndexOutOfBound);
      void make(void) throws(_EXCEPTION_VectorIndexOutOfBound);

public:

      Heap(int m = 32, int n = 32);
      Heap<T,P>& operator=(Heap<T,P>&)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      Heap<T,P>& operator+=(Heap<T,P>&)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      Heap<T,P>& operator-(void)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      int operator+(void);
      boolean operator%(P) throws(_EXCEPTION_VectorIndexOutOfBound);

      boolean operator%(const T&)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      Heap<T,P>& operator<<(const T&)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      Heap<T,P>& operator+(const T&)
            throws(_EXCEPTION_VectorIndexOutOfBound);

      const T& top(void)
            throws(_EXCEPTION_EmptyContainer,
                  _EXCEPTION_VectorIndexOutOfBound);

      T Head(void)
            throws(_EXCEPTION_EmptyContainer,
                  _EXCEPTION_VectorIndexOutOfBound);
```

```
        T Tail(void)
              throws(_EXCEPTION_EmptyContainer,
               _EXCEPTION_VectorIndexOutOfBound);

        Heap<T,P>& operator>>(const T&)
              throws(_EXCEPTION_VectorIndexOutOfBound);

        List<T>* operator>>(P) throws(_EXCEPTION_VectorIndexOutOfBound);
end;
```

In order to describe the methods, let H be a Heap, e an item in the Heap, and k a priority.

-H makes the Heap H empty: `operator-(void)`.
+H returns number of elements in H: `operator+(void)`.
H % k returns true only if an element with priority k is in the Heap H.
H % e returns true only when e is an item in H.
H << e inserts e at head of list of its priority.
H + e appends e to tail of list of its priority.

H >> k returns a **pointer** to the list of all items with priority k. This is useful for checking all items of a given priority, or changing value of an item of a given priority. However, any **removal or addition** of items to this list will **destroy the structure** of the Heap. The second example of this section illustrates the use of this **operator**.

H >> e removes the item e from H.

The methods top(), Head() and Tail() are related to the semantics of Heap. These methods act on the next item of highest priority.

top() returns a **constant** reference to the item of highest priority.
Head() returns the item of highest priority at head of list, and removes it from Heap.
Tail() returns the item of highest priority at tail of list, and removes it from Heap.

Depending on whether you insert (**operator<<**) or append (**operator+**) to the Heap, and whether you retrieve via Head() or Tail(), the Heap will act as a **priority queue** or as a **priority stack**.

```
// Example.zpp

#include<Heap.h>

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------- //

struct element
      int n;
      double b;
```

```
	element(void);
	element(const element&);
	element(int, double);
	operator int(void) const;
	friend outStream& operator<<(outStream&, const element&);
end;

element::element(void)
	n = 47;
	b = 3.5;
end;

element::element(const element& e)
	n = e.n;
	b = e.b;
end;

element::element(int i, double d)
	n = i;
	b = d;
end;

element::operator int(void)
	return n;
end;

outStream& operator<<(outStream& o, const element& e)
	o << "Member n is : " << e.n <<'\n';
	o << "Member b is : " << e.b <<'\n';
	return o;
end;

//////////////////////////////////////////////////////////////////////
// ---------------------------------------------------------------- //
//////////////////////////////////////////////////////////////////////

entry void main(void)
throws(_EXCEPTION_VectorIndexOutOfBound, _EXCEPTION_EmptyContainer)

	using namespace ztlHeapContainerSpace;

	output << "Hello World!\n";

	Heap<element, int> HeapInt;

	element e1(1, 3.5); // high priority
	element e2(2, 7.1); // lower priority
	element e3(1, 9.7); // repeated priority

	HeapInt << e1; // insert at top
	HeapInt << e2; // insert at top of a lower priority
	HeapInt + e3;  // append to tail of the higher priority

	output << "Number of items in Heap is: " << +HeapInt << '\n';

	output << HeapInt.Head() << HeapInt.Tail() << HeapInt.top();
```

```
// Head() and Tail() remove items they return.

      output << "Now again, number of items in Heap is: " <<
                +HeapInt << '\n';

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Number of items in Heap is: 3
Member n is : 1
Member b is : 3.5
Member n is : 1
Member b is : 9.7
Member n is : 2
Member b is : 7.1
Now again, number of items in Heap is: 1
Good-bye World!
```

The following example illustrates the use of **operator>>**, when called on a priority.

```
// Example.zpp

#include<Heap.h>
using namespace ztlHeapContainerSpace;

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //

struct element
      int n;
      double b;

      element(void);
      element(const element&);
      element(int, double);
      operator int(void) const;
      friend outStream& operator<<(outStream&, const element&);
end;

element::element(void)
      n = 47;
      b = 3.5;
end;

element::element(const element& e)
      n = e.n;
      b = e.b;
end;
```

```
element::element(int i, double d)
      n = i;
      b = d;
end;

element::operator int(void)
      return n;
end;

outStream& operator<<(outStream& o, const element& e)
      o << "Member n is : " << e.n <<'\n';
      o << "Member b is : " << e.b <<'\n';
      return o;
end;

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)
throws(_EXCEPTION_VectorIndexOutOfBound, _EXCEPTION_EmptyContainer)


      output << "Hello World!\n";

      Heap<element, int> HeapInt;

      int pr = 3;

// Insert a few items in Heap.

      HeapInt << element(pr, 3.5) << element(pr, 7.1) <<
                 element(pr, 9.7);

// Change value of an item of priority pr in Heap.

      for (Iterator<element> I = HeapInt >> pr; I; I++)
            if (I.Current().b == 3.5)
                  I.Object().b = 4.7;
                  break;
            endif;
      endfor;

// Print values of all items of priority pr in Heap.

      for (Iterator<element> I = HeapInt >> pr; I; I++)
            output << I.Current();
      endfor;

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
```

```
Member n is : 3
Member b is : 9.7
Member n is : 3
Member b is : 7.1
Member n is : 3
Member b is : 4.7
Good-bye World!
```

Appendix VI. Parameter Default Initilizers

Parameter default initilizers can be literals or objects. Once a parameter receives a default initializer, **all** parameters after that must also have initializers.

A constructor for a type is considered a literal.

A function pointer type does not accept default initializers, and in fact does not accept names for its parameters. A function pointer may point to different functions in a context. These functions may or may not have default initializers for their parameters, and when they do, the initializers could be different. Since, parameter default initializers are not factors in determining the type of a function, a function pointer type cannot include that information as part of its definition.

Parameters passed by **reference** cannot receive default initializers.

The following example illustrates these points.

```
// Example.zpp

#include<iostream.h>
using namespace ioSpace;

// ------------------------------------------------------------------ //
// A simple case with literals as default initializers.

void funOne(int i = 25, double d = 2.5)
      output << "Entered funOne...\n";
      output << i << '\n';
      output << d << '\n';
end;

// ------------------------------------------------------------------ //
// An object, like gg below, can be an initializer.

long gg = 77;

void funTwo(long n = gg, short t = 7)
      output << "Entered funTwo...\n";
      output << n << '\n';
      output << t << '\n';
end;

// ------------------------------------------------------------------ //
// When all parameters of a constructor have default initializers
// the constructor become the default constructor, with void signature.
// However, you can supply arguments and override the initializers.

struct argument
      int i;

      argument(int n = 99); // default constructor
```

```
end;

argument::argument(int n)
      i = n;
end;

// ------------------------------------------------------------------ //

struct secondArg
      double d;

      secondArg(void);
end;

secondArg::secondArg(void)
      d = 9.9;
end;

// ------------------------------------------------------------------ //
// Default constructors as parameter default initializers.

void funThree(argument a = argument(), secondArg s = secondArg())
      output << "Entered funThree...\n";
      output << "a.i is: " << a.i << '\n';
      output << "s.d is: " << s.d << '\n';
end;

// ------------------------------------------------------------------ //
// In funFour(), we use a none-default constructor as literal
// for parameter default initializer.

struct thirdArgument
      int i;
      double d;

      thirdArgument(int, double);
end;

thirdArgument::thirdArgument(int n, double b)
      i = n;
      d = b;
end;

// ------------------------------------------------------------------ //
// If function has prototype, initializers must appear in prototype
// not the definition. That holds for methods.

void funFour(thirdArgument t = thirdArgument(47, 97.47));

void funFour(thirdArgument t)
      output << "Entered funFour...\n";
      output << "t.i is: " << t.i << '\n';
      output << "t.d is: " << t.d << '\n';
end;

// ------------------------------------------------------------------ //
// Type function pointer does not take initializers.
```

```
type void funFourPointer(thirdArgument);

////////////////////////////////////////////////////////////////////////
// ------------------------------------------------------------------ //
////////////////////////////////////////////////////////////////////////

entry void main(void)

      output << "Hello World!\n";

      funOne();
      funTwo();
      funThree();
      funFour();

      funFourPointer ffPtr = funFour;
      ffPtr(thirdArgument(777, 77.77));

      output << "Good-bye World!\n";

end;
```

The output of this example is as follows.

```
Hello World!
Entered funOne...
25
2.5
Entered funTwo...
77
7
Entered funThree...
a.i is: 99
s.d is: 9.9
Entered funFour...
t.i is: 47
t.d is: 97.47
Entered funFour...
t.i is: 777
t.d is: 77.77
Good-bye World!
```