Introduction

The ZHMICRO Software Development Platform is based on the concept of "Write Once, Run Anywhere". Specifically, there will always be exactly one Z++ language, independent of an application's targeted Operating System.

The Z++ development language is an established language and has been in its final form for several years. It is based on C++, which is familiar to a large number of developers and includes enhancements that add additional features to modernize the language. This reference manual illustrates these enhancements and has been designed to assist a C++ engineer in becoming proficient in Z++ development. The reference manual is subject to update as further illustrations of the language's capabilities are added for Z++ developers.

Chapter I: Provides an overview of many of the key features of Z++.
Chapter II: Briefly illustrates the graphical user interface of Z++.
Chapter III: Presents greatly enhanced namespace notion of Z++.
Chapter IV: Introduces Invariants and Constraints.
Chapter V: Discusses Z++ enhancement to C++ templates, and provides a reference to Z++ Template Library.
Chapter VI: Discusses Z++ extensions to enumeration type.
Chapter VII: Presents Z++ exceptions with resumption.
Chapter VIII: Is on Z++ unique mechanism for signaling.
Chapter IX: Introduces Z++ tasks for object-oriented threading.
Chapter X: Illustrates Global threading.
Chapter XI: Presents Z++ unique type constructor, collection.
Chapter XII: Provides an introduction to Z++ built-in database SQL statements with object-oriented features.
Chapter XIII: Discusses debugging facilities.
Chapter XIV: Illustrates critical section and mutex facilities.
Chapter XV: Discusses preprocessor commands.
Chapter XVI: Illustrates Z++ operators.
Chapter XVII: Explains Autonomous Agents.
Chapter XVIII: Explains Communicating Concurrent Processes.
Chapter XIX: Explains Asynchronous Function Call.

Chapter I. Language Overview

Chapter X. Global Threads

Chapter XI. Collections

Chapter XII. Database Reference

Chapter XIII. Debugging Facilities

Chapter XIV. Atomic (Critical) Section

Chapter XV. Preprocessor Commands

Chapter I. Language Overview

Section I.1 Introduction

Chapter one illustrates Z++ language for a C++ engineer. It is intended to prepare a C++ engineer for making effective use of Z++ in solving automation problems.

Almost all features of Z++ are covered in the following sections, along with some code examples. Thus, after the first read, this chapter can be used as a quick refresher.

Structure of a program

A filename for a Z++ source program must have extension "zpp". Include, or header files retain their traditional extension, namely "h".

The output of compiler will have extension "zxe". This is equivalent to an "exe" or executable file except Z++ executables run on the Z47 Processor.

A Z++ program must have at least one entry point. This is done by specifying a global function as an entry point, as is done in the following example.

```
entry void main(void)
      //body
end;
```

Besides the main entry, there can be any number of entry points with any signature.

A Z++ executable can be invoked directly, or can be called by other executables. That is, **there is no distinction such as "EXE", "DLL" etc**. It follows that a Z++ program is also a component that can be used in composing larger programs.

The type of return object and the arguments to an entry point can be any valid Z++ type, including user-defined types via the class mechanism. This fact, together with the capability of remote invocation makes distributed computing a lot simpler and abstract. An object is sent over the wire by simply calling an entry point, without any coding. Moreover, what goes over the wire is precisely as you see it in your program, as an instance of some class.

When a program is invoked directly (for instance when program name is double-clicked), the loader picks the **main entry point**.

Execution

The execution begins with initialization of global objects, and then enters the code of the selected entry point. When the body of the entry point ends, global objects are destroyed and control returns to the Z47 Processor.

When a program is invoked as a module by another program, the initialization and destruction of global objects are not repeated for each call to an entry point. This is illustrated in Section I.8 on modules.

Input/Output and Files

For console input and output, include the system header file <iostream.h>, just as you do for C++. In fact, everything works exactly as it does in C++, except the cin and cout objects are named "**input**" and "**output**". Here is a simple example.

```
include<iostream.h>
using namespace ioSpace;
int k = 5;
double d = 7.5;
output << "The value of k is: " << k << " and d is: " << d << '\n';
```

For files, include the system header file <stream.h>. All C++ library functions are included, very much the same way, only cleaned up and simplified. The include file is well documented.

```
include<stream.h>
using namespace fileSpace;
```

Section I.2 Fundamental Types

In addition to all C++ built-in types, Z++ includes **string** and **boolean**. Instead of the word "unsigned", Z++ uses the following naming: **uchar, ushort, uint, ulong**.

Assignments between strings and null-terminated arrays are handled by the compiler, with appropriate warnings whenever applicable. The include files string.h and char.h provide many useful library functions. However, the following operations are predefined for strings.

```
a += b; //concatenate b to a
a = b + c; // a becomes concatenation of b and c
int k = size(a); // k becomes length of a
```

The predefined objects True and False are instances of type **boolean**. Note that language is case-sensitive. Here is an example, using objects True and False to initialize two **boolean** objects, isNew and isOld. You cannot assign 1 or 0 to **boolean** objects.

```
boolean isNew = True, isOld = False;
```

Instances of fundamental types, the ones introduced in this section and enumeration, are in fact objects. Other types are defined via mechanisms like class, using the fundamental types as building blocks.

Enumeration

Z++ makes **enum** more useful and flexible. Consider the following.

```
enum fruits {_apple, _orange};
enum MoreFruits : fruits {_banana};
```

Now `MoreFruits` is really `{_apple, _orange, _banana}`. Although this is not a derivation, we still refer to `fruits` as base type of `MoreFruits`. Note the following.

1. Only one type can appear after the colon, like `fruits` in this example.
2. The extended type must have at least one value of its own, such as `_banana`.
3. The values of extended type follow those of its base type. That is, `_banana` comes after `_orange`.

Unlike C++, the values of an **enum** type, like `_apple`, are not global. This means, **you can reuse them for defining other enum types**. **Also note that the values of enum must begin with an underscore. Other identifier, like names of objects and types, cannot begin with an underscore**.

Z++ provides direct support for many interesting operations on **enum** types, as well as instances of those types. Here we mention the predecessor/successor operators and leave the rest to another chapter.

As in C++, **enum** values can be initialized with integer values. Consider the following.

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

The `++` operator is the successor functions for instances of enums, and the `--` operator is the predecessor function. In the following, `AnotherNumber` is initialized with `_third.`

```
MyNumbers AnotherNumber = ++Number;
```

Note that, `MyNumbers` is an enumeration type, `Number` is an instance of that type, and the strings within braces, like `_second`, are enumeration literals (for a specific enumeration type). Enumeration literals can be initialized with integers. After that, they cannot be assigned to, or compared to integers. We shall see how the bracket function retrieves integer values associated with enumeration literals later.

Section I.3 Control Structures: Selection

Z++ does not need braces {} for initial scope of control structures. You may use them for nested scopes as you do in C++. All control structures have their own closing tag, such as **endif**, **endwhile** etc.

If … else statement

Nested levels of `if … else` can be done as in C++. However, Z++ also supports the elegant approach of Ada, via "**elsif**".

```
if (a > b)
      //statements
elsif (a == b)
      //statements
// ----- more elsif -----
else
      //statements
endif;
```

Note that, **the above example is all one single statement**. The shortest if-statement will look like the following.

```
if (a < b )
      //statements
endif;
```

That is, the closing tag endif serves the same purpose as a pair of braces.

All C++ conditional operators are supported with exact semantics. However, later we will look at Z++ extensions and the expressiveness they provide.

Switch Statement

Z++ does not use "break" for ending each case. On the other hand, Z++ extends C++ by making each **case** a scope. That is, you can declare objects in **case** legs, and they will be local to the leg in which they were declared. Consider the following.

```
switch(expression)
case v1:
      //scope
case v2, v2, v4: //sequence
      //scope
case v5 .. v6: //range from v5 to v6, inclusive
      //scope
else //not required, same as C++ default
      //scope
endswitch;
```

Like other control structures, the braces for **switch** are replaced with its closing tag, namely, **endswitch**. Also, the word "default" is replaced with **else**. Note that, Z++ does not use **break** to end cases. **Thus, if a case does not have any statements and at run time the control goes to that case, nothing will happen**. The control will simply leave the **switch** statement without entering cases below it.

The Ada sequence and range are much more elegant than the C++ repetitive style. When values are in a range, like from 100 to 250, then **100..250** is a lot more concise than repeating 150 cases. The operator for range is two consecutive periods.

When values are such that you cannot put them in a range, then separating them with comma is still more readable than making one case per value. This is called sequence in the above example.

String Literals and Switch

Since **string** is a fundamental type, **switch case** labels can also be **string** literals. Here is a simple example.

```
string animal = "Dog";

switch(animal)
  case "Cat": output << "Saw a cat.\n";
  case "Mouse": output << "Saw a mouse.\n";
  else output << "Saw a dog.\n";
endswitch;
```

The Switch Initial Segment

The section between the **switch** and the first **case** can be used for declaring objects global to all case legs. In fact, any code can appear in this section.

```
switch(argument)
      // Initial segment can contain any code
case label: //first case
      //other cases
endswitch;
```

Conditional Statement

The conditional statement is same as C++ except it returns an object, as opposed to the literal value of an object. It can simply be used the same way as one uses it in C++. The following example shows what it means to return an object. It is not intended as a suggestion for its use, though one may find it useful in conjunction with operator overloading.

```
int j = 5, k = 7;
int r = ++(j < k ? j : k); //r will be 6
```

Section I.4 Control Structures: Iteration

Basically, the pair of braces is replaced with a closing tag. The **continue** and **break** work exactly the same way as they do in C++.

The for-loop is identical to C++ in semantics, as well as syntax except for the tag **endfor**.

```
for (initialization; condition; increments, etc)
      //statements
endfor;
```

The closing tag for while-loop is **endwhile**. A while-loop will iterate as long as its condition remains true.

```
while (condition)
      //statements
endwhile;
```

The do-loop takes **enddo** for its closing tag. Furthermore, its condition is optional and when not present, it becomes a simple infinite loop.

```
do
      //statements
enddo (condition); // or just enddo;
```

Remark. As the choice of words "end do" suggests, a do-loop will end exactly when the condition becomes true. Otherwise put, the loop will iterate so long as the condition remains false.

Section I.5 Unions, Structures and Classes

All mechanism for defining new types, like class as well as functions, use the closing tag **end** instead of a pair of braces. However, within the body of a function, you can use braces with exact semantics as in C++. Generally, you would do that for declaring new objects for local use within the scope of the braces.

Unions

The type constructor **union** is same as C++, allowing methods and operator overloading. As in C++, derivation does not apply to union types. The general form is as follows.

```
union type_name
//members
//methods
end;
```

Classes and Structures

The difference between **struct** and **class** is same as in C++. Thus, elements of a **struct** are **public** by default, while elements of a **class** are **private** by default. We shall use the term **class** to refer to both notions of class and struct.

```
class MyClass
      //members and methods
end;
```

Derivation is multiple-inheritance. Unlike C++, the **default for derivation is public**. Thus, in the following example, `YourClass` is publicly derived from `MyClass`.

```
class YourClass : MyClass
      //members and methods
end;
```

The following is a list of simplifications over C++, without loss of functionality.

1. Z++ does not have the keyword **virtual**. The Z++ compiler is responsible for deciding when to generate code for late (run-time) biding for polymorphism.
2. Instead of C++ pointer **this,** Z++ uses Smalltalk keyword **self**. However, **self** is a reference to object (as opposed to a pointer to object).
3. The overloading of operators is same as in C++.
4. Within the body of definition of a **class**, only prototypes of methods are allowed.
5. All method specifications such as **inline, const, cast** etc. are only allowed for prototypes. The definition of a method neither needs nor is allowed to have any specification.
6. A class cannot have a member that is a reference to another object.

7. Nested type definitions are not allowed. That is, no type definitions can appear inside a **class**.
8. Methods of a class cannot return addresses of, or references to none-public members, unless the return is specified as **const**. It is possible however to remove the **const** via explicit cast.

Note that the restriction mentioned in 7 does not impede expressing elegant solutions. This is less significant than the fact that unlike Ada C++ does not allow nesting the definition of one function inside another. In most cases both of these nesting features are in fact inelegant and programmers seem to go out of their way in order to find ways to make them useful.

It is also inelegant to use a **class** as a **namespace** for hiding other types, such as enumeration. Class is a type constructor and must be used for defining new types. Namespace is a packaging notion, and Z++ extends this notion quite a bit, as we shall see later.

Private Member Visibility

Often, one needs to be able to see the value of a none-public member without changing it. The C++ solution is to write methods that return the value of such members. In Z++ **private** and **protected** members can be made **visible**. The value of a **visible** member can be read, but not modified. Following example illustrates visibility.

```
class Visible
    int x<visible>;
    int y;
public:
    Visible(void);
end;

Visible V;
int k = V.x; // Fine, member x is visible
k = V.y; // Error, member y is not visible
```

Special Methods

Similar to C++, the Z++ compiler provides a default constructor, a copy constructor and a destructor, which can be redefined by programmer. Z++ provides three more special methods. These methods are overloaded operators for assignment, equality and inequality. The prototypes for the six special methods are as follows. Assume the name of class is X.

```
X(void);  //implicit constructor
~X(void);  //destructor
X(const X&);  //copy constructor
X& operator=(const X&);  //assignment
boolean operator==(const X&) const;  //equality
boolean operator!=(const X&) const;  //inequality
```

The default constructor has **void** signature, the signature for copy constructor is the type of class, and the signature for an explicit constructor cannot include the type of class itself.

An explicit constructor taking exactly one argument is called a **conversion constructor**. Unless specified as cast (discussed later in this section), the compiler will in fact invoke a conversion constructor implicitly.

There are only two cases where one can list constructors for members and bases after the colon, the explicit and the default constructors. There are no limitations in such listing for an explicit constructor. That is, one can list any constructor including the default and copy constructors, as well explicit constructors for the creation of members and bases. The other case is the default constructor. The list cannot include anything other than default constructors except for **const** members. Therefore, its only use is for the fundamental types, in particular **const** members. Note that **const** members of class type may be initialized via any one of their constructors. Obviously, only default constructors can be called on bases, which is what the compiler does anyway. Specifically, the copy constructor will always call all necessary copy constructors, and the **programmer cannot change this by listing other kinds of constructors as one can do in C++**.

The three special operators call their respective operators on all bases and members. For instance, the assignment operator calls assignment operators on members and bases. This is in contrast to C++, which implicitly considers the state of an object comprising of state of its members only. **In Z++ the state of an object consists of the states of its members and bases**.

An engineer may never have to define the inequality operator. This is because the compiler defines it as the negation of the equality operator. The compiler defaults for assignment and equality are member-wise. Nevertheless, in presence of pointer members fewer precautions than C++ are needed, as we shall explain in a moment.

Compiler-generated special functions handle pointer members on behalf of engineer. The default constructor initializes pointers to NULL. The copy constructor checks each pointer member of its argument object. If it is null, it simply sets the corresponding pointer to null. Otherwise, it will initialize the member by doing a **new** to it. The object being pointed to is initialized using the object being pointed to by the corresponding member of the argument. **In other words, a copy constructor is in fact what its name says it is**. The case of assignment operator is similar to the copy constructor with one extra step. If the pointer member of the object being assigned to is not null, the **delete** operator is called on it (by the compiler) before copying.

The default for comparison operator compares pointer members as pointers, not the objects being pointed to. This is the meaning of the state of an object.

Other Extensions

In addition to the C++ operator `->` for reaching members, Z++ provides the operator `->>` for reaching bases.

The specification c**const** for methods is same as C++. In addition the keyword `cast` can be used for constructors and conversion operators. Consider the following example.

```
class X
      int k;
public:
      X(int) cast;
      operator int (void) const, cast;
end;
```

Now compiler will require explicit cast for going from **int** to X and backwards.

Section I.6 Templates and Casting

Z++ templates are full-fledged C++ templates, with simplification and correction. Casting is as general as it can be accomplished by several ANSI additions to C++. However, a **single simple syntax does it all**. The language Z++ is designed and crafted from scratch. It includes all of C++ independent of platform, and with corrections. Moreover, Z++ extends C++ for **safe and abstract distributed computing**.

Templates

Instead of the term class, Z++ uses the term type, as in "`template<type X>`". Other differences are in reduction of C++ syntax requirements.

Z++ does not entirely delay the compilation of templates until after parameter substitution. Furthermore, it is not possible to instantiate template parameters with objects or numeric literals as C++ compilers allow. The parameters can only be instantiated with previously defined types.

Casting

Explicit conversions are done via cast. The syntax is as follows (cast returns an object).

```
cast(result_type, object);
```

The argument object is the instance to be cast, and result-type is the type of object we wish to cast to. In general, cast creates the object it returns. An exception would be when a conversion operator returns a reference to another object.

The specification of being a constant applies to objects, not types. One can specify the object resulting from a cast to be a constant, or not a constant for that matter, by including or excluding the **const**. Thus, in the following the resulting object will be const.

```
cast(const result-type, object);
```

Now, if object was in fact const and we do not specify const for the result, then the **const** of object is simply **cast** away.

The cast can also be used to navigate bases of an object pointed to. ANSI has added several forms of cast to C++ to accomplish the same thing.

The C form of casting is also supported for pointers, as shown below.

```
int* p;
void* v = (void*) p;
```

As in C++, a constructor can be used to cast from a previously defined type. For instance one can write `result_type(x)` to convert x to an instance of `result_type`, when there is such a constructor. However, the **compiler does such conversions implicitly**, unless the constructor is specified for explicit cast.

Conversion operator |-

For conversion between string and numeric type, the conversion operator replaces all library functions usually supplied with C++ compilers. Consider the following line.

```
Left |- Right;
```

The symbol for conversion operator is the OR "|" symbol followed by the minus sign "-". The rules are simple. One of the operands, Left or Right, must be of type **string**, and the other must be of numeric type. Numeric types are **char**, **short**, **int**, **long**, **float**, **double** and their unsigned equivalents.

The operator always converts its Right operand to its Left operand. Thus, if Left is of type string, the conversion is from numeric to ASCII. On the other hand, if Right is of type string, the conversion is from ASCII to numeric. The operator also returns the result of conversion, so it can be used as arguments to calls, assignments etc.

Section I.7 Threads

Multi-threaded programs in Z++ are abstract and safe. The Z47 Processor takes the entire responsibility for creating, removing threads, queuing calls and blocking callers. To create a thread, simply replace the word **class** with **task**, as shown below.

```
task MyTask
      //private section
public: //methods for task interface
      void taskMethod(void);
end;
```

The set of public methods of a task makes up its interface. Calls to public methods will be queued and serviced in the order in which they arrive. The caller will block until the call returns, although Z++ provides a none-blocking mechanism as well.

Each task appearing as a base or a member will have its own thread.

Since a **task** can have task members, and can be derived from other tasks, an instance of a **task** could be multithreaded.

Consider the following illustration using the above example.

```
{ //start a scope
      MyTask HardWork;     //thread is created here
      HardWork.taskMethod();   //caller is blocked
} // HardWork and its thread are destroyed here
```

At the point of declaration, `HardWork` is created with its own thread. On the next line we are making a call to a method of `HardWork`. At that point, the call is queued for `HardWork`, and the thread making the call is blocked. When `HardWork` receives the message `taskMethod()`, and services the request, the call returns, and caller is unblocked.

When an instance of a **task** is created dynamically via the **new operator**, the threads associated with the dynamic instance will be destroyed when the object is deleted.

Section I.8 Modules and Components

A Z++ program is also a module, a DLL, a package or a reusable component.

Conceptually, a module is an identifiable part of software with well-defined boundaries in the form of a so-called contract. The module can be modified (therefore replaced) so long as the boundary conditions are not affected. One says that the module hides information, meaning that the module can only be dealt with as a black box.

Z++ executables can be used as part of a larger program. Thus, Z++ is designed for distributed component-oriented (platform-independent) development.

The set of entry points of a module constitutes its boundary. Thus, the entry points of a module are the only points through which it can be called from outside of the module. The (invocation) state of a module is the combined state of its global objects before the execution of the body of an entry point begins. When a module is used as a component, it may be invoked multiple times. It is essential to know exactly when the state of an invoked module is initialized. The protocol followed by Z++ is described in the next section.

Module Invocation

By module invocation we mean making a call to an entry point of a module from within another module at execution time. In order to invoke a module, one first introduces the module as a type in the calling module. Consider the following example.

```
class MyModule = "MyCode.zxe"
      //private section
public:
      external int module_method1(long, string&);
      external string& module_method2(int);
end;

MyModule MyInstance;
```

In this example, `MyCode.zxe` is **the module being introduced**. The type introducing the module is called `MyModule`. The introduction can be done via type-constructors class or struct. For multi-threading one can use **task** instead. The methods specified as **`external`** are entry points of `MyCode.zxe` module.

Note that **`external`** is not the same as **`extern`**, which is used for external linkage of objects exactly the same way as it is used in C++.

*MyModule is a class in an ordinary sense and can have other kinds of methods besides those specified as **external**.*

When a module has been introduced, an instance of its type is said to represent the module. Thus, the object `MyInstance` represents the module `MyCode.zxe`.

Recall that the state of a module is the combined state of its global objects. **The state of a module persists for the life of the object representing it**. The state of its module is initialized when the object is created. This simply means that all global objects of the module are created and initialized in accordance to their constructors.

Each invocation of the module (calling one of its entry points) may modify its state. The modifications remain in effect until next invocation.

The module is finally removed when the object representing it goes out of scope. At that point all global objects of the module are destroyed.

Keep in mind that, a dynamic object created via the **new operator** will exist independent of scope in which it was created, until explicitly deleted. The lifetime of a module represented by an object depends on the object.

The Z47 Processor has far more capabilities than what we have mentioned here. As one would expect, a module may be invoked via a URL across the Internet. Thus, a module may be loaded and executed on local machine, or otherwise, the remote machine may execute the code of the module on behalf of the local machine.

When a URL to a remote server is used, the default action is to download the module to the local machine and execute it as a process of the local Z47 Processor. In the following, however, the remote module will execute as a process of the remote Z47 Processor (that is, on the server).

```
class MyModule = "URL to reach MyCode.zxe"<remote>
```

The specification **`remote`** can be changed to the default action, **`local`**. A remote URL is indicated with "`ZPP://ip-adress/`". What follows the header portion is the full path to reach the module. For instance, "`C:/top/…/MyCode.zxe`" is an example for Windows.

Z++ can also use dynamic libraries of other languages, like C++ DLLs, as modules. We will discuss this in connection with Service-oriented Architecture. Furthermore, class invariants and method constraints can also be used in component-oriented development, as we will illustrate in a later chapter.

Section I.9 Exceptions

Z++ extends the traditional approach to handling exceptions.

- The Z47 processor raises a predefined set of exceptions, like division by zero.
- Users can extend the set for their own needs.
- When an exception occurs, one can either **resume** or **repeat**.

Exception values are enumerations. The system exceptions are the enumeration values in **enum** exceptionEventType, defined in system header file exception.h. To define your own exceptions just extend the set as shown in section on Enumeration.

The linguistic construct for exception takes the following syntax.

```
include<exception.h>
using namespace exceptionSpace;

layer<exception_type>
    //body of code within layer
handler
    case some_exception:
        //service code
    else //catchall, if needed
       //service code
endlayer;
```

You may wish to handle different sets of exceptions at various levels. You specify the intended set via <exception_type>, where exception_type is just the **enum** type name of the set of exceptions you wish to handle.

The keyword **layer** marks the start of a level in your code where you wish to handle exceptions. If any exception is raised within the layer, the program execution goes to its **handler** section. If handler section has a case that services the raised exception, program execution will continue from after **endlayer**. Otherwise, it will continue to go to the next layer until either a handler is found, or the Z47 Processor level is reached. When execution reaches the Z47 Processor level without being serviced, your program will be terminated. **The scenario just described assumes you are not using resumption**.

Here is how you raise your own exceptions.

```
raise some_exception;
```

The Z++ exception mechanism hides a great deal of complexity. Consider the case where you invoke a module from inside a layer, and that the exception occurs in one of the threads of that module. While we cannot describe the operational semantics in this introductory chapter, it is good to know that the system takes care of you in the most desirable and meaningful manner.

The cases in handler section can take ranges and sequences of values as illustrated for the switch statement.

There are two resumption mechanisms, **`repeat`** and **`resume`**. Whenever you can actually repair the cause of an exception use the **`repeat`** statement to go back to the statement that caused the exception. This allows retrying the operation once the problem has been fixed. On the other hand, **`resume`** takes the control back to the statement following the one that caused the exception. Generally, **`resume`** is for cases when the execution can proceed without fixing the problem that caused the exception, but that one needs to inform the user, or log the problem.

Section I.10 Revisiting Enumeration

Recall that enumeration literals can be initialized with integer values. Here, we wish to see how to access those integer values, as well as certain literals of an enumeration type.

The mechanism provided by Z++ is called the **bracket function**. Consider the following.

```
enum someDays {_Sunday, _Monday, _Friday};
someDays Today = _Monday;
```

Here, the compiler will initialize `_Sunday` with 0, then the following literals with 1 and 2. So the integer value of `Today` is 1. Consider the following.

```
int Value = [Today];
```

In the above example, the integer object `Value` will be initialized with 1. Thus, the bracket function applied to an enumeration **instance** retrieves its associated integer value.

For iteration, one would like to initialize an object with the first literal of an enumeration type, and iterate until all literals are visited. When the bracket function is applied to an enumeration **type**, it returns the first **enum** literal for that type. To get the last **enum** literal of an enumeration type, use **double brackets**. Let us illustrate this with an example.

```
enum X {_one, _two, _three, _infinity};
for (X counter = [X]; counter <= [[X]]; counter++)
endfor;
```

The bracket function `[X]` returns the first literal for the type, in this case the literal `_one`. This initializes the object `counter`. On the other hand, `[[X]]` in the conditional segment of the for-loop returns the last literal, i.e. `_infinity`. This loop will execute exactly four times. Note that the operator ++ is the successor function for enumeration types.

Section I.11 Logical and Comparison Operators

The logical operators of C++ are usually called short-circuit. Z++ retains the exact semantics as in C++, but provides more operators for complete circuit evaluation. Consider the following code fragment.

```
if (X && Y)
```

When the operand X is false, we already know the whole condition will evaluate to false, even if the operand Y is true. We may choose to evaluate Y anyway, or simply skip its evaluation. This can change the state of the program when the operand Y is an expression, or a function call. The choice made in C (and thus C++) is to skip the evaluation of Y, thus the name short-circuit. C++ treats the operator `||` analogously.

In Z++ one can force the evaluation of operands. The full-circuit operator for logical **and** is `<>` and the one for logical **or** is `><`. In the following, both operands X and Y will be evaluated.

```
if (X <> Y) //full-circuit and
if (X >< Y) //full-circuit or
```

Z++ also provides an operator for logical **exclusive or**.

```
if (X ^^ Y) //true exactly when one operand is true
```

Obviously, you can mix and match all logical operators, as you need.

The arithmetic operator `^^`, and its pair `^^=`, when applied to numeric operands will perform exponentiation.

```
int j = 2;
j ^^= 3; // j is 8 now
```

Quite often, we need to make a chain of conditions as in the following.

```
if ((a < b) && (b <= c) && (c == d))
```

Z++ provides shorthand for this chain. You can use the following, instead.

```
if (a < b <= c == d)
```

Section I.12 Arrays: operator overloading

Z++ extends all numeric operations to numeric arrays. Here are some examples.

```
int a[5][7];

//your code to set values for cells of array a

long b[5][7] = a; //initialize b using a
short s[5][7] = 47;  //initialize all cells with 47

//Below, first increment every element of s.
//Then set every cell of b equal to the
//product of corresponding cells of s and a.

b = ++s * a;

b += a; //apply operator+=() cell-wise
```

In short, whatever you can do with instances of numbers, you can also do with instances of compatible numeric arrays.

Z++ extends overloaded operators for base-type to the array as a whole. Here is an example.

```
struct base
//members, methods
      void operator+(int); //overload
end;

base a[15];
a + 23; //apply operator+(int) to every cell
```

Compatible arrays can be compared for equality and inequality. Let us also mention one case of declaring arrays of classes. Consider the following.

```
class_type array[7](a, b, c);
```

Here, a constructor of `class_type` requires three arguments, like the objects a, b and c. The constructor is applied to every cell of array. **In contrast, C++ allows only the use of default constructor**.

Section I. 13 Dynamic Arrays

The correct terminology is semi-dynamic arrays in that the sizes of dimensions are not known at compile time. However, after elaboration the sizes cannot be changed. Consider the following example.

```
int DA[m][n];
```

The indices m and n are some integer objects, and therefore their values are not known at compile time.

The term static is not used in Z++. Here, by a static array we mean one for which the compiler knows the sizes of its dimensions. For instance, the following is a static array, which is allocated on the heap instead of the stack.

```
typedef integerTwenty int[20];
integerTwenty* SAP = new int[20](97);
```

Now SAP points to an array of 20 integers, all initialized to 97. The important thing is that, SAP is a static array because the compiler knows the size 20 of its dimension. The following is an equivalent definition using dynamic array, where m is an integer object.

```
typedef dynIntegers int[];
dynIntegers* DAP = new int[m](97);
```

It is convenient to get the size of each dimension of a dynamic array without having to keep a copy of it. This is useful in constructing loops, among other things. The operator **size** does just that.

```
int a[m][n]; //two-dimensional dynamic array
int first = size(a, 0); //size of first dimension
int second = size(a, 1); //size of second dimension
```

Instead of literals 0 and 1 any discrete numeric object can be used for dimension (the second operand for size).

Section I.14 Degenerate Array Pointers

Along with adding dynamic arrays to Z++, the language expressiveness must be adjusted to avoid unintended incorrect statements without limiting their use. The type degenerate array can only be defined via **typedef**, as follows.

```
typedef degenIntArray int[];
```

By itself degenIntArray is of no use. Note, however, that it contains the information for base type, in this case **int**, and that the array is one-dimensional dynamic array. A pointer of type degenerate array is the only kind of pointer that is allowed to point to a dynamic array of the same base type and same number of dimensions. Consider the following, where m is an integer object.

```
int* p = new int[m]; //unlike C++ this is an error in Z++
degenIntArray* q = new int[m]; //this is fine
degenIntArray* r = new long[m]; //incorrect base types
```

Note that the degenerate pointer q can point to any one-dimensional array of base type **int**, which is the reason for nomenclature. Furthermore, a degenerate pointer cannot point to a static array. Instead, one must do the following.

```
typedef IntArrayPointer int[20]*;
IntArrayPointer p = new int[20];
```

That is, for a static array the pointer must be of exact type, including the size of each dimension.

Let us consider the use of templates, typedef and dynamic arrays in combination. Suppose you define the following template class.

```
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;
```

You can instantiate a dynamic array of Base without using degenerate pointers, like this. First we define couple of integers.

```
int m = 7;
int n = 3;

Base<int, double> B[n](m, 3.7);
```

Since n is three, this will create an array with three cell of instantiated type of Base and will initialize them by calling the constructor on each of them. You can use it as usual, for instance as follows.

```
B[1].show();
```

However, if we want to create the same array on the heap, we would do something like the following. First define a **typedef**.

```
template<type U, type V> typedef MyBase Base<U, V>[];
```

The name of **typedef** is MyBase and its type is a degenerate template. We can use the **typedef** MyBase as shown below.

```
MyBase<int, double>* MBP = new Base<int, double>[n](m, 3.7);
```

On the left, we are instantiating the **typedef** MyBase and turning it into a pointer, because the **new** operator on the right will return a pointer. On the right, we are defining a dynamic array, as was done a few lines earlier. We can use this array like this.

```
(*MBP)[1].show();
```

First we follow the pointer MBP to reach the array, then we use the index to reach a particular cell (1 in this case), and finally we make the call to the method show().

We close this section with a simple example.

```
typedef MyArray double[][]*;
MyArray MAP = new double[m][n]; //m, n are integer objects
MyArray p = MAP;
MAP = new double[m+=5][++n];
```

Section I.15 Global Threads

Object-oriented threading in Z++ is abstracted in the form of **task type constructor**, presented in Threads. There are times, as in some cases of client server models, that **global threads** provide much better abstraction.

Consider the following.

```
void globalFun(double)<thread>;
```

This is a plain global function, specified as thread. The semantics are what one would expect. At the point of call, the global function becomes a thread and lives on its own. **The thread that made the call is not blocked**.

Threads associated with instances of tasks are terminated when the instances are destroyed. The question now is, how is a global thread terminated?

The Z++ abstraction is in the form of a mechanism for **Inter-Thread Communication**. The basic idea is that, one thread sends a system-wide signal, and the intended thread catches the signal. So now we need to know how to generate and catch signals.

The base type for signals is signalEventType defined in Z++ system header file, exception.h. Let us extend it with a new signal, `_SIGNAL_TerminateYourself`.

```
enum MyOwnSignals : signalEventType {
      _SIGNAL_TerminateYourself
};
```

A system-wide signal is sent as shown below, using the reserved word **`signal`**. We are sending the signal `_SIGNAL_TerminateYourself` to Z47 Processor. The operator **`<-`** generates signals and events.

```
signal <- _SIGNAL_TerminateYourself;
```

This statement will presumably appear in the thread that created the global thread, or at any point that we wish to tell the global thread to terminate itself. Then, at some point in the body of globalFun(), where we wish to end the thread, we do the following.

```
do enddo(signal ? _SIGNAL_TerminateYourself);
```

Recall that do-loop does not exit until the condition for **`enddo`** becomes true. The above line will probably appear as the last line in the body of the global function. If so, the global thread will simply wait until the signal arrives, at which time the condition of **`enddo`** becomes true. Then it leaves the loop, thereby terminating the thread.

The question **?** operator checks system signals for a match. If Z47 Processor finds one, the result of the expression will be true, and Z47 Processor will remove the signal from the system queue. Otherwise, the expression evaluates to false.

Inter-Thread Communication

It is important to realize that signaling is an asynchronous mechanism for inter-thread communication. Create your custom signal values by extending the system signals or a set derived from it, as was done here. Then, use them to make the intended threads execute blocks of their code based on the signals that they receive. We will discuss this further in a later chapter.

Section I.16 Common (static) Members

The C++ notion of a static member of a class is called **common** in Z++. The use of term common is due to the fact that a **common member** of a class will have the same literal (value) for all instances of that class.

The declaration of a common member has the following pattern.

```
common int sharedMember : setCommon(int), incCommon(void);
```

In the above line, `common` is the keyword (like C++ static), **int** is the type of member named `sharedMember`. Following the colon is the list of methods (of the class) that are **authorized** to modify `sharedMember`. The list is given in the form of prototypes without the return type. Thus, in above example, `setCommon()` and `incCommon()` are the only methods that can modify `sharedMember`. All other methods, including constructors/destructor can access `sharedMember`, but cannot change it in any way.

The colon, and the list of **authorized methods** following it are optional. **A common member cannot be public**. Thus, the only way to modify a **common** member is to invoke one of its authorized methods.

A common member is initialized same way as a C++ static member is.

```
int Class_name::sharedMember = 7;
```

The semantics are identical to C++ in that the initialization takes place prior to the execution of your program's entry point.

In C++, one can define a class consisting of entirely static methods. That turns the class into no more than a global namespace. In languages that lack global scope, like Java, this is an effective technique for creating global objects. C++ does not really need that technique, and should be avoided.

In Z++, a common member cannot be manipulated until an instance of the class owning it has been created. Then, authorized methods can modify it as desired.

Section I. 17 Namespaces

Z++ generalizes the notion of C++ **namespace** providing more expressiveness. Among the features are the following.

- Namespaces can be derived same way as classes can.
- Namespaces can have **private**/**public** sections, just like classes.
- One can separate a namespace definition from its implementation.
- The opening of a namespace can be ended.

The following is an example of declaring a **namespace**.

```
protected namespace ExportBase

int BaseInt = 49;

struct BaseStruct
      double dbl;

      BaseStruct(double);
      double get(void);
end;

BaseStruct bsInstance(77.79);

endspace;
```

The specification **protected** for a **namespace** is optional. If present, the **namespace** can only be used as a base for derivation by another **namespace**.

Although the definitions of methods for `BaseStruct` can be given here, we will show how Z++ also allows the implementations to be separated, so they can be done in a separate file. Let us first define another **namespace** deriving from `ExportBase`.

```
namespace ExportSpace : ExportBase

class SpaceClass
      string name;
public:
      SpaceClass(const string&);
      string get(void);
end;

SpaceClass scInstance("Hello World!");

endspace;
```

The implementations shown below can be done in the same file, or a separate file.

```
implementation ExportBase
```

```
BaseStruct::BaseStruct(double d)
      dbl = d;
end;

double BaseStruct::get(void)
      return dbl;
end;

endspace;

implementation ExportSpace

SpaceClass::SpaceClass(const string& s)
      name = s;
end;

string SpaceClass::get(void)
      return name;
end;

endspace;
```

The **using** statements are same as C++, except they can be ended. For instance, consider the following statement, which **exports the public section** of the **namespace**.

```
using namespace ExportSpace;
```

One can end the above exportation with the following statement.

```
endusing namespace ExportSpace;
```

The two equivalent statements for exportation of a specific item are as follows.

```
using ExportSpace::ExportBase::bsInstance;
endusing ExportSpace::ExportBase::bsInstance;
```

Note that `bsInstance` cannot be accessed directly because its **namespace** was specified as **protected**. Since default derivation is **public**, we can reach it via `ExportSpace`.

All other C++ features, like renaming a **namespace**, can be done via derivation, and with more control.

Section I.18 Class Invariants

The most important feature of an object is that it maintains its own state. As things are, the checking of the state of an object is scattered in the methods of the class defining the type of the object. **Invariants localize the conditions for desirable state of the object, and are checked transparently**. Furthermore, invariants are inherited in derivations.

Let us look at an example and then follow it with explanations.

```
class MyClass
  int a;
  double d;

  invariant(a > 50) trigger();
  invariant(d < a) _some_Exception;

protected:

  void trigger(void);

public:

  MyClass(void);
  void setValues(int, double);

end;
```

An invariant is a **boolean expression** among the members of a class. The semantics is that, the condition of an invariant must hold at all times. Otherwise, when the condition becomes false, the specified action will take place. **The action could be raising an exception, or invoking another method, perhaps to repair the damage**.

In the above example, violations of the first invariant will invoke the method `trigger()` while violations of the second invariant will raise the specified exception.

Invariants are tested at end of every public method, except the destructor. In the above example the two invariants will be tested at end of the constructor, and the method `setValues(int, double)`, transparent to your code.

Note that none-public methods cannot be invoked without having to go through a **public** method. Therefore, the **invariant** are only tested for **public** methods. The test is performed at end of each method, just before returning from the method.

Section I.19 Method Constraints

Constraints are conditions that must hold prior to the execution of the code of a method. Constraints are also known as contracts. Consider the following example. Explanations will follow.

```
class MyClass

  int a;

protected:

  void trigger(int);

public:

  long doSomething(int b)
  {
    (b == 0) trigger(b);
    (b < a) _someException;
  };

end;
```

The constraints for a method are specified as part of its prototype. A constraint is a **boolean** expression followed by an action, which can be raising an exception or invoking another method. **Constraints are tested before the execution of the body of a method**. The test is transparent to your code.

The semantics is that, when the condition of a constraint becomes false the specified action will take place. The violation of the first constraint for the method `doSomething(int b)` will invoke the method `trigger(int)` and violation of the second constraint will raise the specified exception.

Section I.20 Collections

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 **class** types.

In order to define a **collection**, we need an enumeration 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.

```
class Shape
    //members, methods
end;

// First value

class Square : Shape
    //members, methods
end;

// Second value

class Rectangle : Shape
    //members, methods
end;

// Third value

class Triangle : Shape
    //members, methods
end;
```

Here is how the **collection** is defined. Explanations will follow.

```
collection basicFiguresType<basicFigures> {
            _square<Square>,
            _rectangle<Rectangle>,
            _triangle<Triangle>

      void Initialize(void);
      void PrintAreas(void);

};
```

First notice that a **collection** can have methods. In fact, collections can be derived from one another, as we will show in a later chapter.

The name of **collection** type is `basicFiguresType` and `basicFigures` is its associated enumeration type that we defined earlier. Then, to each literal of the enumeration we are associating a type. For instance, `Square` corresponds to the enumeration literal `_square` and so on.

Recall the use of bracket functions and operators `++` and `--` for successor and predecessor functions for the enumeration type. They also work on **collection** type. In fact, an instance of **collection** maintains an implicit tag, which can be reached, and reset via the bracket function. Consider the following definition for the body of the method of the **collection** defined above.

```
void basicFiguresType::PrintAreas(void)
  output << "Area at current tag.\n";
  switch([self])
    case _square:
      output << "Area of square is : " << self[_square].area() << '\n';
    case _rectangle:
      output << "Area of rectangle is : " << self[_rectangle].area() << '\n';
    case _triangle:
      output << "Area of triangle is : " << self[_triangle].area() << '\n';
  endswitch;
end;
```

The argument of **switch** evaluates to the current value of collection's implicit tag.

The **bracket function** on enumeration literals returns their numeric value, as before. Thus, to reach a value in an instance of a **collection** we use the familiar array notation. For instance, `self[_square]` evaluates to the instance of Square corresponding to the literal `_square`. At that point, we can call any of the methods of type Square, and we have chosen to call `area()`.

The implicit tag allows using a **collection** like a tagged union. However, in case of **collection** the objects do not occupy the same space.

Collections have more features, such as **shared** methods, which reduce the coding effort as in the definition of `PrintAreas()`. We will illustrate these features in a later chapter.

Section I.21 Travel Statement

The Z++ travel statement is an abstraction for the notion of strong mobility. An agent is a Z++ component, which includes one or more travel statements.

The travel statement can appear in any context in an **entry point** of an agent. In order to avoid obscure programs, the travel statement may only appear in an entry function. Unrestricted jumps from any point in a large component will create a situation like the wild use of go-to statement.

The semantics is that, upon execution of a **travel** statement, the agent terminates itself at the local node. The agent then begins execution with the statement following the **travel** statement, at the destination node. That is, the state of the agent is transferred along to the destination node.

A Z++ traveling agent is a process of the distributed operating system Z47.

The **travel** statement can appear in any context, including in nested exception layers. When the **travel** statement raises an exception, the agent does not leave the local node. Thus, one can handle the exception, and perhaps resume or even repeat the **travel** statement. Below is an example of use of **travel** statement in an exception **layer**.

```
layer<exceptionEventType>

      travel URL_string;   // go to the IP address

handler

      case _EXCEPTION_SendError:
            // Change URL_string and try again
            repeat; // try the travel statement again

endlayer;
```

The execution of **travel** statement may raise the following exceptions.

```
_EXCEPTION_ConnectError.
_EXCEPTION_MemoryException.
_EXCEPTION_SendError.
_EXCEPTION_ReceiveError.
```

A connect exception occurs when server does not accept the request to send the agent to it, or whenever connection cannot be established. The memory exception occurs when server does not have sufficient memory to allocate space for the agent's executable. The send and receive exceptions may occur due to communication errors.

Chapter II. Introduction to Graphical User Interface

Section II.1 Canvas

Z++ GUI presentations are created with tools. In this section we look at what the tools generate, and their rather straightforward meaning.

The screen used by the tools to create GUI is called a **canvas**. Once the **canvas** is prepared, the tool generates an include file for inclusion in your source. The output of the tool is also called a **canvas**, and looks like the following example.

```
canvas Variety
      button Done;
      button Next;

      label State_Name;

      checkbox One;
      checkbox Two;

      radiobutton First;
      radiobutton Second;

      combobox States;

      field Text;
end;
```

The term **canvas** is a Z++ reserved word. The syntax of **canvas** is similar to that of structure or class, without methods. The name of our example **canvas** is Variety.

The terms like **button**, **label** etc. are not Z++ reserved words. They only have a meaning as graphical entities of a **canvas**. The strings following them are the names you chose for them when creating the **canvas** with tools. Those names will appear on the entities whenever applicable. For instance, the button named Done will result in a button labeled Done, and the label for checkbox One will be `One`.

The more interesting fact is the convenience with which you can reference entities of a canvas, without having to deal with global C-define values. In Z++ you simply use the names you have chosen, like Done, One etc in the same way you use **namespaces**. For instance, `Variety::Done`, or `Variety::Text`. We shall illustrate this mechanism in a later section in this chapter.

Note that the entire **canvas** as you see here is generated, by tools. You simply include it in your source files like any other header file. The tools make it easy, and quite intuitive to recognize and use the graphic entities.

Section II.2 Frame

As we saw, a **canvas** is generated, by tools. A **frame** is a type constructor identical to the type constructor **class**, specifically for manipulating a **canvas**.

In order to use and manipulate a **canvas**, we associate it to a **frame**. Here is an example. Note that **frame** is a keyword like **class**.

```
frame Complex := Variety

      string value;
      boolean mark;
      int index;

public:
      Complex(void);
      $Complex(interfaceEventType&);
end;
```

The association operator := associates the **canvas** Variety to **frame** Complex. Since a **frame** is same as a **class**, all rules hold. You can have other frames as members of a frame, etc. Indeed, the significant difference is the association to a **canvas**, and the instinct method, which we will illustrate shortly.

The Z++ system include file interface.h contains all the definitions of events and types used in GUI presentations. We present some of them here for convenience.

```
enum interfaceEventSignals {
      _IES_Draw_Signal,
      _IES_Erase_Signal,
      _IES_Pen_Tap_Signal,
      // other events ...
      _IES_Invalid_Signal
};

struct interfaceEventType
      ushort x;
      ushort y;
      ushort entity
      interfaceEventSignals Event
end;
```

The members of structure `interfaceEventType` have the following meanings.

The x is the horizontal coordinate of a point (pen, mouse) from left of **canvas**, and y is its vertical coordinate from top of **canvas**. As we shall see shortly, the Z47 Processor sets the values of all the members.

The member entity will be the name of graphic entity on which the pointer landed. For instance if the button named Done was tapped, then entity will be Done. The member Event will be one of the enumeration literals of type `interfaceEventSignals`.

Instinct Method of a frame

The instinct method is declared similar to the destructor but uses $ (the dollar sign). The type of argument to instinct method is `interfaceEventType`, which is passed by reference. It is the responsibility of the Z47 Processor to invoke the instinct method, and to pass the argument to it. As far as your code is concerned, the **frame** does that by instinct, thus the nomenclature.

Obviously the **switch** statement is quite useful in the body of an instinct method. However, recall that names of graphic entities are not global-defines. Thus, apart from **switch**, we need a mechanism similar to a **switch**, which can have names of entities for its **case** labels. That is precisely what a **select statement** does. Here is an example of the body of an instinct method. The actions are omitted so you can see the structure.

```
Complex::$Complex(interfaceEventType& e)

  switch(e.Event)

    case _IES_Draw_Signal:
    case _IES_Erase_Signal:

    case _IES_Mouse_Click_Signal:

      select(e.entity) //select statement starts here

        case Done:
        case One:
        case States:

      endselect; //select ends here

  endswitch;
end;
```

In general, you **switch** on the Event. For some events, such as tapping the screen with a pointer, you use **select** on the entity member of the argument.

Once again, the Z47 Processor sets the values of members of the argument e. It is also the responsibility of the Z47 Processor to invoke the instinct method and pass the argument to it. **You simply focus on your logic**.

Section II.3 GUI operations

Z++ extends C++ familiar operations to GUI elements, **without regard to platform**. The appearance and coloring of GUI entities may look different on different devices. However, there is no Z++ statement specific to any platform. Thus, your Z++ programs mean one and the same thing for all platforms.

Now we illustrate the following example. From previous section you recall that it is the instinct method of a **frame** called Complex, which was associated to the **canvas** called Variety.

```
Complex::$Complex(interfaceEventType& e)

  switch(e.Event)

    case _IES_Draw_Signal:

      Variety::Text << value; //print Hello World (the string value)

// Populate the combobox States

      Variety::States << "Texas";
      Variety::States << "Colorado";
      Variety::States << "New Mexico";
      Variety::States << "Kansas";
      Variety::States << "Iowa";
      Variety::States << "South Carolina";

      Variety::One << True; //set checkbox One as checked

      return; // not needed

    case _IES_Erase_Signal:
      $self; //erase operator can only appear in instinct
      return; // not needed

    case _IES_Pen_Tap_Signal:

      select(e.entity) //select statement starts here

        case Next:

        case Done: //tell canvas to erase itself
          Variety <- _IES_Erase_Signal;

        case Two:
          ~Variety::Text; //clear field
          $Variety::States; //toggle visibility

      endselect;

  endswitch;
end;
```

The Z47 Processor sends the signal `_IES_Draw_Signal` as soon as it completes the drawing of the canvas. The purpose is to perform any initialization, such as populating lists. In our example, we assume the constructor of the **frame** has initialized the **string** value to "Hello World". The C++ output (insertion) operator has been extended to GUI entities. Thus, the string "Hello World" will be printed to the field Text of **canvas**. Recall that Text is name of a field of **canvas** Variety.

In a similar manner, we populate the list of States. This can be combined with getting the strings from a file.

Let us see how a **canvas** is erased. Once a **canvas** is erased, the instinct method of the associated **frame** **will not be invoked**, and will not receive events. That is as it should be.

In this example, we have decided to end the session, and erase the **canvas**, when user taps the button named Done. Looking at the case label of **select** statement for this button, we see the Z++ statement: "`Variety <- _IES_Erase_Signal;`". The operator `<-` sends signals. In this case, we are sending the erase signal to the **canvas**. Well then, we must receive that signal, as we do. We catch the event in the **switch** case labeled `_IES_Erase_Signal`. Now is a good place to present the simple Z++ mechanism for drawing and erasing a **canvas**.

The operator for drawing and erasing is just the **`$`** symbol used for the instinct method. Notice the logical behavior of Z++ with regard to a **frame**. When you declare an instance of a **frame**, as in "`Complex MyFrame;`", only the **frame** constructor is invoked to create and initialize the object. Nothing is drawn at this point. To draw the **canvas** associated with this **frame**, just apply the draw operator to the instance, `"$MyFrame;"` and you are done. As we saw earlier, the Z47 Processor will generate the draw signal for you so your code for GUI initialization will be executed.

The erase process is just as simple and logical. Recall that **`self`** is a reference to object itself, and is equivalent to the C++ **`this`** pointer. To erase a **canvas**, you tell its **frame** to do so for you, like this `"$self;"`.

Graphical User Interface Operators

In this section we list Z++ GUI operators for reference. All these operators are platform independent.

`operator <<`

This operator has been conveniently overloaded with rather obvious semantics, for various GUI elements as discussed below. Note that the compiler checks on the validity of statements involving these operators, so you do not have to worry about doing the right thing for every detail.

For a checkbox or a radio-button, only a **boolean** object or value can be output. If true is output the entity will be set as checked, otherwise it will be unset (unchecked).

For a field a **string** is output, which appends to the content of the field. For replacing the field contents with new value use operator = (assign).

For a combo-box a discrete numeric is output, which sets its current selection.

For a label or button, a **string** is output, which replaces their text.

operator >>

The extraction operator is just the opposite of insertion operator. Thus, it is used for receiving input from GUI objects.

For a checkbox or a radio-button a **boolean** is returned. True indicates the entity is checked while false indicates that entity is unchecked.

For a field its content is returned in a **string**.
For a combo-box the index of its current selection is returned.
For a label or button their text is returned in a **string**.

operator $

This is the draw/erase toggle **operator**, when applied to GUI entities in a canvas, in addition to its use for drawing or erasing the canvas itself. When applied to canvas entities, it will make them visible or invisible. For instance, `$Variety::States;` will either make the combo-box States visible, or it will hide it.

operator ?

The question operator returns a **boolean**. It will return true if the object is showing, otherwise it will return false. Consider the following statement.

```
if (!?Variety::One) $Variety::First; endif;
```

This reads as follows. If the checkbox named "One" is not drawn (not showing), toggle the visibility of the radio-button called "First".

operator ~

When applied to a field, this operator clears the contents of the field. Note that the contents are lost.

When applied to a combo-box, the contents of the list of the combo-box are deleted.

operator <-

In the context of GUI operations, the **signal operator** is used to send events to a canvas. For instance, one sends the erase signal to a canvas.

operator <<-

This **operator** sends its signal to the canvas that is the parent of the canvas to which it is applied.

operator +

For a combo-box, this operator adds its string operand to the list of the combo-box.

operator -

For a combo-box, this operator removes its string operand from the list of the combo-box.

operator =

For a field, this operator takes a string. The string operand replaces the contents of the field. Note that operator << appends its string operand to the field content.

For a combo-box, the **string** operand is what will show in its field area. This string is not inserted in the list of the combo-box.

For a checkbox or radio-button, the string operand will show for their label. Note that the value (label) of a button or label (static control) is changed via **operator** <<.

operator @

The focus operator is a prefix operator applied as in "`@entity`". It sets the focus to the entity to which it is applied.

operator &

The get-focus operator is a prefix operator applied as in "`&entity`". It the entity has focus it returns true. Otherwise it returns false.

operator ^

Reports whether the entity is enabled or disabled (grayed out). This is a prefix operator, as in "`^entity`", and returns **boolean**.

operator !

The enable/disable operator is a prefix operator. Disabling may either gray out the entity, or hide it, depending on the capabilities of the underlying system. It is also a toggle operator. Thus, if entity is already disabled, "`!entity`" will enable the entity.

operator []

This is a convenient operator for getting the value of an item in the list of a combo-box, or to change its value. The integer (discrete numeric) value for the bracket is the index of the string in the list we wish to reach. The syntax is similar to that of an array where the combo-box identifier is used instead of the array. You can assign a string, or receive a string using this operator.

operator ? string

This is a binary operator for getting the index of a string in the list of a combo-box. If the string is not the list of the combo-box it returns –1.

operator size

Operator size is a Z++ operator for getting length of a string or sizes of dimensions of a dynamic array. When applied to a combo-box it returns the size of its list, i.e. the number of elements in the list of the combo-box.

Message Boxes

Three kinds of GUI message boxes are currently available: Information, Confirmation and Error style. The following enumeration is in the Z++ system header file, interface.h.

```
enum interfaceMessageBoxType {
      _MBT_Invalid_Kind,
      _MBT_Single_Button,
      _MBT_Double_Button,
      _MBT_Triple_Button
};
```

The number of buttons tells the Z47 Processor what type of buttons to draw. A single button will show "OK". A double button will show two buttons, "Yes" and "No". A triple button will also show a third button, "Cancel".

Suppose, `MyCanvas` is name of a canvas, and response is an instance of the type **short**. Consider the following line.

```
response = MyCanvas << "Do you wish to continue?" ? _MBT_Triple_Button;
```

This will draw a confirmation box, with three buttons. The value of the button tapped by user, 0, 1 or 2 will be stored in the object response. It is the question mark just before the number of buttons that will draw a confirmation box. A colon will draw an information box, and the exclamation symbol "!" will draw and error box.

```
MyConvas << "Error Reading file." ! _MBT_Single_Button;
MyCanvas << "Transmission Complete." : _MBT_Single_Button;
```

Message boxes are generally used to inform user of some event, and may also ask for a decision by user. The mechanism should be simple and involve as few parts as possible.

The compiler checks for correct combination of style and number of buttons. For instance, logically speaking, an information box can only show an "OK" button. So the compiler will complain about the use of double or triple buttons for an information box. This is better than simply providing a mechanism and requiring the engineer to remember everything else.

Chapter III. Namespace Reference

It is desirable to choose a name, for a type or an object alike, 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.

Some languages prohibit the use of global space. Ironically, a class treats its members as global objects within its scope. The methods of the class manipulate its members directly and without being passed to them as arguments. **Somehow this seems reasonable even when a class is quite large**.

In a language equipped with the capability of linking separately compiled files in a project, such as Z++, the notion of **extern** allows the sharing of an object among several files. Z++ **namespace** is identical to class except for not being able to create an instance of a **namespace** as one can with a class. This approach facilitates interesting designs where the **public** global functions of a namespace behave essentially the same way as methods do for a class.

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 to their members through their methods**.

Section III.1 Namespace Sections

Namespaces have **private**, **protected** and **public** sections with same semantics as with 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** 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 definitions 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.

Section III.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 definition 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 keyword **endspace** closes the definition of a namespace, as well as its implementation.

```
implementation ExportSpace

      SpaceClass::SpaceClass(string s)
            name = s;
      end;

      string SpaceClass::get(void)
            return name;
      end;

endspace;
```

Section III.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.

Section III.4 Protected Namespaces

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 users to access their contents.

To solve the above problem, Z++ namespaces can be specified **protected**.

```
protected namespace Name-of-namespace
// body of namespace
endspace;
```

The semantics is that, a **protected namespace** can only be derived from. In particular, users cannot directly access the contents of a protected namespace. Since the derivation could be **private**, the contents of a protected namespace will remain hidden from users of the library.

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.

Section III.5 The 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 exported 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;
```

Exported **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 IV. Invariants and Constraints

Exception mechanism is a sophisticated linguistic construct. Nonetheless, the notions of invariants and constraints are not syntactic sugars for what can be accomplished through exception handling techniques. Furthermore, the value of these notions in developing reliable software has been demonstrated pragmatically.

Section IV.1 Invariants

The state of an object can be modified through its **public** methods. Often, it is desirable to ensure that the state of an object remains in an acceptable range. The **invariant** mechanism allows the specification of a safe range of state for an object, as well as, the actions to be taken should the state of the object fall outside of the specified range.

The safe range of state for an instance of a class is specified with **boolean** expressions, also known as class invariants. The Z++ **invariant** mechanism provides two possible actions in response to the violation of a class invariant. One can either raise an exception, or invoke a **private** method, which we refer to as a trigger.

Class invariants can appear in any section of the definition of a class, such as **public** or **private**, because they are handled internally by instances of that class. The **boolean** expressions defining the class invariants must use the (data) members of the class. Moreover, invariants are inherited.

Class invariants are tested at end of execution of its **public methods**. This ensures that the execution of the method did not leave the object in an unacceptable state. When an invariant is violated (its expression becomes false), its specified action will be triggered.

The syntax of **invariant** mechanism is as follows.

```
invariant (boolean expression) exception-name;
invariant (boolean expression) method-call;
```

The keyword **invariant** is followed by the invariant's expression, and then followed by either an exception to be raised, or a private method to be invoked in response to the violation of the **invariant**.

The following examples illustrate the use of invariants. We begin with defining a few exceptions by extending the Z++ system exceptions.

```
#include<exception.h>
using namespace exceptionSpace;

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

Now we define a class with invariants.

```
struct baseInvariantClass
      double d;
      short s;

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

      baseInvariantClass(void);
end;
```

Next we define a **class** with invariants. Note that this **class** will inherit the invariants of the above **struct**.

```
struct derivedInvariantClass : baseInvariantClass

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

      int a, b;

private:
      void trigger(void);
end;
```

The following class, plain, will inherit all the invariants of its bases.

```
class plain : derivedInvariantClass
end;
```

Section IV.3 Constraints

Constraints are conditions that must hold prior to the execution of an invoked **public** method of an instance of a class. For instance, incoming arguments may have to satisfy certain conditions, which may or may not involve the members of the class. Method constraints are also known as contracts.

The Z++ constraint mechanism is similar to the invariant mechanism without the use of the keyword **invariant**. That is, a constraint is specified with a **boolean** expression for the condition of the constraint, followed with an action. The action could be raising an exception, or calling a **private** method of the class. Method constraints are specified at method prototype, in the body of a **class**. Constraints are enclosed between braces {}.

As with invariants, the violation of a constraints occurs when its condition becomes false. However, constraints are part of the definition of a method and are not related to inheritance. **This means that a new definition for a method in a derived class must define its own constraints, if any**.

The following example illustrates the use of constraints. The exceptions used in this example are the same ones defined for the example on invariants, in previous section.

```
class constraintType
      int i;
      double d;

      void trigger(double);

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;
```

Chapter V. Template Reference

Template is the Z++ linguistic mechanism for expressing **Abstract Data Types** (ADT). Since an ADT is a type, **template** is a type constructor. An ADT is the behavioral abstraction of a type for composing more complex algorithms. The template mechanism assigns the task of implementing an ADT for a particular use, to the compiler.

A specific implementation of an ADT is called an instantiation. A Z++ template is a parameterized type, which accepts previously defined **types** as arguments and produces an instantiation for the ADT that it defines.

Often, it is desirable to specialize an ADT for a particular context. This is somewhat different than the mathematical specialization of the notion of a group to a commutative group or a ring. In terms of programming, we are interested in instantiating a template with types that posses certain behavior. The Z++ linguistic mechanism for this concept is **template pattern**, which restricts the class of types that can instantiate a **template**.

The Z++ template mechanism supports **class invariants** and **method constraints**.

Section V.1 Defining a template

The definition of a **template** is similar to that of a **class** or **task**. In particular, an instantiation of a **task template** is a **task,** and is therefore threaded. In fact, the type instantiating a template parameter can be a **task**.

Below is an example of a **template** definition, which we will use in the next section for illustrating template **pattern**.

```
template<type T> 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;
```

Section V.2 Template Pattern

Template **pattern** is a Z++ linguistic mechanism for specifying permissible types for instantiating templates. For instance, looking at the example in previous section, you will notice that the **template** is only useful when instantiated with fundamental types like **int** or **string**. Although the example is simple for illustration purposes, it is easy to see the significance of **template pattern** in designing correct programs.

There are two ways for specifying patterns. We either list previously defined types as possible candidates for instantiating a **template**, or we may specify the methods that an instantiating type must posses. We will give an example for each case, starting with the former (i.e. listing previously defined types).

```
pattern template<T> StringPattern
        T : string;
end;
```

The name of this pattern is `StringPattern`, and it specifies one type only, **string**. More types can be added using comma as a separator. We can also specify each type on a separate line, just like the type **string**. Finally, **we can specify any previously defined type, including user-defined types**.

Now, specifying a pattern for instantiating the parameter of a template is quite easy. For instance, the header of the **template** we defined in the previous section takes the following form.

```
template<type T : StringPattern> class StringType
```

That is, the template parameter T is followed with a colon and the name of the pattern. Obviously, when there are several template parameters, you can specify a pattern for each parameter. Pattern specification for each template parameter is **optional**.

Let us define another pattern, in this case specifying methods that the instantiating type must have.

```
pattern template<T> OperationPattern
        operator T(void);
        T operator+=(T);
end;
```

The **pattern** `OperationPattern` specifies two methods and in this case both methods are operators. The first is the familiar conversion operator. Here is an example of a template that uses this pattern.

```
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;
```

The pattern specification means that, if the instantiating type does not define those methods as (**public**) methods, you will receive an error. Actually, `StringType`, the **template** we defined in previous section defines those methods. Below is an example illustrating the use of these templates.

```
entry void main(void)

      StringType<string> stgType("Initialized");

      StringOperations<StringType<string> >
            stgOps(StringType<string>("Hi there."));

// Next line outputs the string "Hi there.Initialized".

      output << stgOps.combine(stgType) << '\n';

end;
```

When specifying methods for patterns we can specify the names of methods, or use the question mark ? for their names. The question mark will match any method-name with the specified signature.

Section V.3 Invariants and Constraints

Templates can have class invariants and method constraints, as usual. The ZTL template library, discussed in next section, uses these notions.

Section V.4 Z++ Template Library (ZTL)

The ZTL template library include files are well documented. In this section, for a start, we provide an overview of namespaces and file dependencies. Future revisions will include more detailed illustrations.

The system include file ztlException.h defines the following exceptions, raised by **ZTL**.

```
namespace ztlExceptionSpace

enum ztlExceptionType : exceptionSpace::exceptionEventType {
      _EXCEPTION_EmptyContainer,
      _EXCEPTION_IncompatibleVectors,
      _EXCEPTION_VectorIndexOutOfBound,
      _EXCEPTION_HashItemNotFound,
      _EXCEPTION_HashItemIndexOutOfBound,
      _EXCEPTION_LastNotUsedContainer
};

endspace;
```

All **ZTL** namespaces are derived from **namespace** `ztlExceptionType`.

Container Base Namespace

ZTL includes the containers **List**, **Stack** and **Queue**. These templates are defined in the system header file Iterator.h, and are used via following declarations.

```
#include<Iterator.h>
using namespace ztlListIteratorSpace;
```

The namespace `ztlListIteratorSpace` is derived from a few namespaces, which we describe here for purposes of documentation.

The specification **protected** disallows the use of the **namespace** except for derivation by other namespaces.

```
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;
```

```
endspace;
```

The template `Template_Element` maintains a copy of element to be inserted in the container, and a link. It only provides implicit (default) and copy constructors. We will use this **template** in defining containers.

Template `Container` is the base for all container Abstract Data Types that use the Iterator template. This template provides some behavior. For instance, it can tell its size i.e. how many items it contains. It can make itself empty, and can also tell if an item is a member. The term membership as used in Set Theory means belonging to the set, or being a member of the set.

The use of these operators is consistent in Z++ Template Library (**ZTL**). The prefix + operator applied to a container returns its size, and a prefix – operator makes the container empty. The operator % is the closest symbol to the Set Theory symbol for belonging.

Finally, we need to make a forward declaration for the notion of iteration.

Template BaseList

The next namespace in hierarchy contains definitions for List, Queue and Stack.

```
namespace ztlListContainerSpace : ztlBaseContainerSpace
```

This namespace first defines internally used **template** BaseList.

```
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;
```

Template `BaseList` derives from template `Container` and has two members, a **`friend`** `Iterator` and the member `last`, a pointer of template type `Template_Element`. Its behavior factors all the general abstract behavior that we think a container such as list should have. However, for each case we will introduce the behavior (interface), using operator overloading, or method names that are more meaningful for the particular ADT.

List

Template **List** is intended for use in your programs. Here is its definition.

```
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);
      boolean operator%(const T&) const;
      List<T>& operator=(const List<T>&);
      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;
```

Operator + returns the number of elements in the list.
Operator – makes the list empty.
Operator % returns true if the element is in the list, otherwise returns false.

The assignment, equality and inequality operators are also overloaded for lists.

Insertions in list take a few forms for convenience. For inserting at head of list, we can either use operator << or method insert. For appending to tail of list we can either use operator + or method append.

Method makeHead tries to find the element and put it at head of list. If it does not find the element in the list, it will insert it at head of list (same as method insert).

Removing an element from list takes two forms, with different semantics. Operator – removes only the first occurrence of an element. On the other hand, operator >> removes all occurrences of the element in the list.

To remove the element at head of list use method Head. Method Tail removes the element at tail of list.

We can reach the elements at head or tail of a list without removing those elements. Method First returns a reference to the element at head of list, while method top returns a constant reference to it. Similarly, Last returns a reference to the element at tail of list, while bottom returns a constant reference to it.

We can extend a list by appending another list to its tail via operator +=.

Queue

Below is the definition of **template Queue**, and explanations follow.

```
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;
```

Operator + returns the number of elements in the queue.
Operator – makes the queue empty.
Operator % returns true if the element is in the queue, otherwise returns false.

The assignment, equality and inequality operators are also overloaded for queues.

An element can be inserted into a queue either via operator << or the method push. Method pop removes the first element inserted, and top returns a constant reference to that element.

We can extend a queue by appending another queue to its tail via operator +=.

Stack

The definition of **template Stack** is similar to that of **template Queue**.

```
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;
```

Operator + returns the number of elements in the stack.
Operator – makes the stack empty.
Operator % returns true if the element is in the stack, otherwise returns false.

The assignment, equality and inequality operators are also overloaded for stacks.

An element can be inserted into a stack either via operator << or the method push. Method pop removes the last element inserted, and top returns a constant reference to that element.

We can extend a stack by appending another stack to its tail via operator +=.

Iterator

In order to use containers **List**, **Stack** and **Queue**, you will need to open the iterator's **namespace**, as shown below.

```
#include<Iterator.h>
using namespace ztlListIteratorSpace;
```

The namespace and **template** Iterator are defined as follows.

```
Namespace ztlListIteratorSpace : ztlListContainerSpace

enum containerTargetKind {_List, _Queue, _Stack};

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); //post ++ operator
      Iterator<T>& operator++(void);
end;

endspace;
```

Vector

Z++ language includes the notion of semi-dynamic arrays where the size of each dimension of an array is not known at compile time. Furthermore, Z++ generalizes the notion of C++ semi-dynamic arrays created via the **new operator** at run-time. Template

Vector is a true dynamic array allowing the array to be resized at run-time. The **Vector template** is included in a program as follows.

```
#include<Vector.h>
using namespace ztlVectorSpace;
```

Vectors are simply zero-based arrays and therefore do not require specialized iterators. Furthermore, the resizing is controlled by, user, as illustrated shortly. The few operators overloaded for Vector make its use simple and convenient.

```
namespace ztlVectorSpace : ztlExceptionSpace

// The direction of typedef is opposite to that of C++.
// degenrateTemplate is the new name for type.

template<type T> typedef degenrateTemplate T[];

// Vector_Element is used internally by template Vector.

template<type T> struct Vector_Element
      degenrateTemplate<T>* array;
      Vector_Element<T>* link;

      Vector_Element(int);
      ~Vector_Element(void);
end;

// ===== This is the template used in your programs =====

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) { // This method has constraint
            (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;

endspace;
```

The constructor for **Vector** **template** has two parameters, with default value of 32. The first parameter is for the initial size of **Vector**. The second parameter is the factor by which the **Vector** will expand or shrink.

An instance of **Vector** can tell its own size via the usual prefix **`operator+`**. Suppose we have constructed a vector (an instance of **Vector**) using default values of 32. This vector will have 32 cells allocated. Now suppose we need to increase its size to 65, using the expansion **`operator<<`**.

```
vector << 65;
```

The above statement will increase the size of vector to 96 cells. Note that the default expansion factor is 32. Thus, 32 + 32 is 64, one less than 65. For that reason, another expansion factor of 32 is needed to reach, or pass 65 cells.

Now suppose, we wish to reduce the size of vector to just 40 cells, using the shrink **`operator>>`**.

```
vector >> 40;
```

This will reduce the size of vector to 64 cells, which is the initial size of 32 plus one expansion factor of 32 in order to reach or pass the number 40.

Template **Vector** also overloads the usual operators for assignment and comparison. The clear **`operator~`** puts the instance back to the way it was at its initial construction.

Array

Z++ language directly supports array operations. As for dynamic arrays created via the **template Vector**, the **Array template** extends Z++ support for arrays to dynamic arrays.

The **template Array** is a restriction of **template Vector** to numeric types. This is done via the following template pattern.

```
pattern template<T> ztlArrayPattern
      T : char, uchar, short, ushort, int, uint, long, ulong;
end;
```

Array overloads all the usual numeric operation, and is included as follows.

```
#include<Array.h>
using namespace ztlArraySpace;
```

Hash Table

The hash table is included in a program using the following two lines.

```
#include<Hash.h>
using namespace ztlHashSpace;
```

However, the **namespace** `ztlHashSpace` is constructed in several steps, which we are going to explain. The first **namespace** is `ztlHashTableSpace` derived as shown below. This **namespace** defines the **template pattern** `ztlHashPattern` for instantiating the **Hash template**. This pattern requires the instantiating type to have a constructor that takes the type **string**, and must overload the conversion **operator**, returning a **string**.

The **namespace** also defines the type of hash function, and it provides a default hash function `zDefaultHasher`. It then defines the **template Hash**, which is the hash table. The actual **namespace** included in a user program is explained after this.

```
namespace ztlHashTableSpace : ztlListIteratorSpace, ztlVectorSpace

pattern template<T> ztlHashPattern
      T(string); // type type must have a c-tor taking string
      operator const string&(void) const;
end;

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;
```

```
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;

endspace;
```

Now we define the **namespace** included in programs. The **namespace** `ztlHashSpace` defines the final piece of construction, namely the hash-table iterator.

```
namespace ztlHashSpace : ztlHashTableSpace

template<type T> class HashIterator

      HashIterator<T>& operator=(const HashIterator<T>&);

      Hash<T>* H;
```

```
        Iterator<T> I;
        int counter;

public:

        HashIterator(void);
        HashIterator(Hash<T>&);
        HashIterator(const HashIterator<T>&);
        HashIterator<T>& operator=(Hash<T>&);
        const T& Current(void);
        T& Object(void);
        operator void*(void);
        HashIterator<T>& operator+@(void); //post ++ operator
        HashIterator<T>& operator++(void);
end;

endspace;
```

Remark. The post ++ **operator** is defined using "+@", but applied as usual using ++.

Heap

Heap, or priority queue, is implemented using dynamic array. Each cell of the array represents a priority, and holds a list of objects for that priority. Thus, for each priority the heap can act either as a queue or a stack depending on the directions of insertion and removal from the list. The **Heap** **template** is included in user program as follows.

```
#include<Heap.h>
using namespace ztlHeapContainerSpace;
```

The definition of this **namespace** is straightforward. The include file is well documented and can be used as a reference until this work is completed.

Chapter VI. Enumerations

Enumeration is a useful type constructor. However, mixing literals of an enumeration type with ordinary identifiers, especially in a large program, becomes confusing. Treating enumeration literals as integers is also a possible source of confusion.

Z++ approach to enumeration alleviates the above confusions, and endows this type constructor with several desirable extensions.

Enumeration Literals

.
Z++ identifiers (names of types and objects, etc.) cannot begin with an underscore. On the other hand, enumeration literals must begin with an underscore, as shown below.

```
enum fruits {_apple, _orange};
```

The default integer value for the first literal is 0, and default value for each following literal is one more than the value of its preceding literal. It is possible to assign other values so long as the following value is larger than the value of its preceding literal.

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

The literals of an enumeration type are private to the type. Therefore the same string can appear as literal for several enumeration types. In case of conflict, a literal can be qualified with its type name, as follows.

```
enum_type_name::enum_literal
```

Integer value of a literal

An enumeration literal cannot be mixed with an integer in an expression. However, its integer value can be retrieved via the **bracket function** []. Below, the integer `intNumber` is initialized to 30.

```
enum MyNumbers {_first = 10, _second = 17, _third = 22};
MyNumbers Number = _second;
int intNumber = [Number] + 13; // 17 + 13 == 30
//int intNumber = [_second] + 13; // same as above
```

Extending Enumerations

Z++ enumerations are extensible, as shown below.

```
enum fruits {_apple, _orange};
enum MoreFruits : fruits {_banana};
```

In this example `fruits` extends `MoreFruits`. That means, `MoreFruits` contains all the literals of `fruits` followed by the new literal `_banana`.

Z++ uses the extensibility of enumeration types as a means for allowing engineers to extend Z++ system exceptions and signals. These will be illustrated in their respective chapters. Collection type constructor, a generalization of enumeration also relies on the ability to extend enumeration types.

Successor and Predecessor Operators

The `++` operator is the successor functions for **enum** objects, and the `--` operator is the predecessor function. In the following, `AnotherNumber` is initialized with literal `_third`.

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

First and Last literals

Often in applying enumerations in our solutions we need to use the first or the last literal of an **enum** type. Using these particular literals directly may require code change, should we decide to modify the **enum** type.

The **bracket function** was introduced earlier in this chapter for retrieving the integer value of an enumeration object or literal. When the bracket function is applied to an enumeration **type** (rather than object) it retrieves the first literal value of the type. In the following example, the object `Today` is initialized with first literal of its type.

```
enum someDays {_Sunday, _Monday, _Friday};
someDays Today = [someDays]; // Today == _Sunday
```

To retrieve the last literal of an enumeration type use **double brackets**.

```
someDays AotherDay = [[someDays]]; // AotherDay == _Friday
```

The bracket function makes it possible to write loops that require less maintenance. Often, one would like to initialize an object with the first literal of an enumeration type, and iterate until all literals of the type are visited. Below is an example of use of bracket function in a for-loop.

```
enum X {_one, _two, _three, _infinity};
for (X counter = [X]; counter <= [[X]]; counter++)
endfor;
```

The bracket function `[X]` returns the first literal of the type, in this case the literal `_one`. This initializes the object `counter`. On the other hand, `[[X]]` in the conditional segment of the for-loop returns the last literal `_infinity`. This loop will execute exactly four times. Note also that the operator `++` is the successor function for enumeration types.

Chapter VII. Exceptions

Exception mechanism was introduced in Section I.9. Here we add a few more details.

First, we need to clarify the issue of exceptions occurring in threads, or remote modules. If such an exception is not handled within the thread (or module), that thread is terminated, but the exception is passed on to its parent. Eventually, if the thread is the top-level parent just before Z47 Processor, and the exception has not been handled yet, the program as a whole will terminate.

A function (or method) can only raise user-defined exceptions listed at its prototype. For instance, consider the following.

```
void fun(…) throws(_userExceptionOne, _usrExceptionTwo);
```

This function fun can only raise the two indicated exceptions in its **throws** list.

Now, any function that calls fun must also include all un-handled exceptions in the **throws** list of fun, in its own **throws** list. The compiler uses this scheme to make sure that all user exceptions that may be raised are **eventually caught**. This eventuality, however, ends at an **entry** point to a module. That is, the compiler will not complain if you do not catch exceptions within an entry point because they may be handled by another module that has invoked that entry point. Nonetheless, the compiler requires you to include all such exceptions in the **throws** list of the entry point, which serves as a reminder.

Note that, if an exception in the **throws** list of a function is caught within that function, you do not need to include it in the list of **throws** of a function calling it.

Chapter VIII. Signals

In Section I.15 we introduced the notion of signals. Briefly, **signal** is a Z++ built-in object with two operators. A signal, like an exception is an enumeration. One introduces new user-defined signals by extending the system signals, as illustrated in Section I.15.

Operator <- generates a signal. For instance, if `_MySignal` is a user-defined signal, the following statement generates that signal.

```
signal <- _MySignal;
```

A generated signal is, maintained by the Z47 Processor, which keeps it until some process or thread accepts the signal. **Operator ?** retrieves a signal as shown below.

```
if (signal ? _MySignal)
  //do something
endif;
```

If the signal `_MySignal` has arrived, it will be removed from Z47 queue, and the **boolean** expression will be true. Otherwise the expression will evaluate to false. Note that, one can also use an object instead of literal signal value of `_MySignal`, for either generating or retrieving signals.

All kinds of signals discussed in following sections are defined in the system header file exceptions.h, with namespace **exceptionSpace**.

Process-bounded Signals

These signals must be derived from **signalEventType**. A process-bounded signal can only be caught within the same **process** that generated it. They are useful for signaling among the threads of a single process.

A signal of this kind will only be caught once. That is, as soon as a thread catches a signal of this kind, Z47 will remove that signal from its waiting list.

Node-bounded Signals

These signals must be derived from **universalEventType**. When a process generates this kind of signal, **any process**, including itself, can catch the signal. **They are intended for signaling among threads of different processes**.

A signal of this kind will only be caught once. That is, as soon as a thread catches a signal of this kind, Z47 will remove that signal from its waiting list.

Process-bounded Entire Signals

The term entire means that a signal of this kind can be caught by the entire set of threads of a process, **except the main thread itself**. That is, when the signal is generated all threads that have registered to be notified will in fact catch that signal. This includes global threads, as well as **task** threads. However, these signals are process-bounded so that they can only be caught **within the process that generated them**.

These signals must be derived from **threadEntireEventType**. Any thread that wishes to catch this kind of signal must register for it. Registration is done via the **accepts**(…) specification as illustrated below.

A global thread registers its bounded entire signals via **accepts**(…) specification in a manner similar to **throws**(…) specification.

```
void MyGloblaThread(void)<thread> accepts(_sig1, _sig2,…);
```

A **task** specifies its accept-list of signals anywhere within its definition, preferably in the **private** section near **invariant** listing. For a **task**, these signals are intended for use as signals for **task handlers**.

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**.

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.

Node-bounded entire signals should not be specified for **entry** points of an application that is intended for use as a module (another application will load it, locally or remotely). This is because the module may receive unwanted signals from other processes, especially when loaded for remote execution.

Tell and Hear Signals

These signals generalize communication among process. The ordinary model is what one usually refers to as Communicating Sequential Processes, which is based on the notion of rendezvous. In this model, the caller blocks until it hears from the process it called, or just gives up. In tell-hear signaling mechanism caller passes data without waiting for a response. The communication model may be referred to as **Concurrent Communicating Processes**.

In this model, a process tells a signal and a list of objects, as in a function call. Another process, local or remote, that has registered to receive the signal, will eventually hear the signal and receive the associated data. Thus, data transfer in this model is more like sending an email, as opposed to making a Remote Procedure Call.

Registering and Receiving Hear Signals

The registration of hear signals is similar to the registration of accept signals, except a hear signal must also list the types of the objects to receive the associated data. Note that the keyword is **hears**, analogous to **throws** and **accepts**.

```
hears(_sig < type_1, type_2, … > , … )
```

The list of types between the angled brackets < and >, as well as the brackets, are optional. Note that the ellipses (three dots) are **not** part of the syntax. A hear signal, upon arrival, will be accepted only if the types of the data arrived match as in a function call. Otherwise, the signal will be rejected and the sender (teller) will be informed to raise exception.

The last set of ellipses following the comma is for more signals.

The acceptance and rejection of a hear signal is transparent to processes. Z47 Processors perform these operations. When a hear signal is accepted, it will be put into the queue of the process that has registered for it. In this case we consider the signal to have arrived.

A hear signal is received by a thread of the process that has registered for it in the same manner as other signals. However, the keyword instead of **signal** is **hear**, and there may be a list of objects.

```
hear ? sig : object_1, object_2, …
```

The semantics of this **boolean** expression is as follows. If the signal sig has not arrived yet, the expression evaluates to false. On the other hand, if signal has arrived, it will evaluate to true, and the following will occur. The signal will be removed from the queue of the process, and the arrived data will be copied to object_1, object_2, etc.

Sending (Telling) Hear Signals

The keyword for generating a hear signal is **tell**, and it takes more optional operands than a regular signal. A hear signal can be sent to a remote process by name (of the process). In addition, a hear signal may include a list of objects to send. Below is the full syntax.

```
tell <- sig $ URL , process : object_1, object_2, … ;
```

The only operand required is the signal sig. The $ symbol is a separator like the comma. Since both operands are optional, the kind of separator distinguishes whether the string is a URL, or the name of the process to receive the signal. The list object_1, object_2, etc. is the data associated with the signal to be delivered along with the signal.

When the URL and process name are not present, but the signal is intended to deliver data, 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, 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 can raise the following exceptions, in which case the receiving end has rejected the acceptance of the signal.

1. `_EXCEPTION_SIGNAL_SignalNotRegistered`. This exception is raised when the receiving node informs the sender (the node on which tell statement was executed) that no process has yet registered for the signal.
2. `_EXCEPTION_SIGNAL_TellOperandTypeMismatch`. In this case, the receiving 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**(…) for this signal.

Remark. In an entire application (running as a process) hear signals are unique. Compiler will not allow the same hear signal to be specified more than once. Hence, exactly one thread in a process can register each hear signal.

Hear expressions can only appear in the same thread that has specified their signals. Therefore, if several tell data arrive before matching hear expressions are executed for a

particular signal, they will be delivered in the order in which the tell statements were executed. No signals will be 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 one that has registered for the signal. Hear signals are not entire, so they do not propagate.

Remark. Tell/hear signaling is for inter-process communication, particularly between processes on different nodes (remote signaling). 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 tell statement does not end too soon. This is so because in case of signaling within a single process, Z47 processor uses references to the data sent along with a tell statement, while in remote signaling, objects are actually copied.

Chapter IX. Object-oriented Threads (tasks)

Z++ provides two abstractions for threading. In this chapter we illustrate the object-oriented abstraction, and in the next chapter we will discuss the global abstraction.

The Z++ type constructor **task** is threaded. The syntax for defining a **task** type is the same as that of **class**. In fact, tasks can be derived with the same semantics as classes, and you can also define **task templates**. Tasks can have **invariants** and **constraints**, just like classes.

The set of **public** methods of a **task** makes up its interface. Calls to **public** methods will be queued and serviced in the order in which they arrive. The caller will block until the call returns (Z++ provides a none-blocking mechanism as well, discussed later).

Since a **task** can have **task** members, and can be derived from other tasks, an instance of a **task** could be multithreaded. This is because **each task appearing as a base or a member will have its own thread.**

The Z47 Processor handles the creation and destruction of threads for a **task** object, as illustrated below.

```
task MyTask
      //private section
public: //methods for task interface
      void taskMethod(void);
end;

{ //start a scope
      MyTask HardWork;     //thread is created here
      HardWork.taskMethod();   //caller is blocked
} // HardWork and its thread are destroyed here
```

At the point of declaration, `HardWork` is created with its own thread. On the next line we are making a call to a method of `HardWork`. At that point, the call is queued for `HardWork`, and the thread making the call is blocked. When `HardWork` receives the message `taskMethod()`, and services the request, the call returns, and the caller is unblocked. When an instance of a **task** is created dynamically via the **new** operator, the threads associated with the dynamic instance will be destroyed when the object is deleted.

Task Idler Method

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 section we look at task idler, and in the next section we discuss task signal handlers.

A task idler is a **private** method, which **cannot be called directly**. Instead, task object will invoke the 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 @.

```
task MyTask
      @MyTask(void); // idler
public:
      MyTask(void);  // constructor
end;
```

The signature of idler is **void**, and it does not return anything. Furthermore, an idler is not inherited. Each task has its own idler.

Task Signal Handlers

Task objects can respond to signals just as they service incoming requests for their public methods. The specification of a handler is straightforward, as shown below.

First, define a set of signals by extending system signals (or an extension of it). A handler is just a **private** method of **task** specified with a signal to handle. The return and the signature of a handler must be **void**.

```
#include<exception.h>
using namespace exceptionSpace;

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

task MyTask
      void FirstHandler(void)<_SIGNAL_FirstIdler>; // A handler
      void SecondHandler(void)<_SIGNAL_SecondIdler>; // another handler
      void ThirdHandler(void)<_SIGNAL_ThirdIdler>; // a third handler
public:
      MyTask(void);  // constructor
end;
```

The definition (body) of a handler does not need the signal specification. As a general rule, Z++ only takes **method** specifications for prototypes, when a type is being defined.

A handler cannot be called directly. Instead, a task object will invoke its handlers when corresponding signals arrive.

Task signal handlers are not inherited. This is because in a derivation each base is also a task and runs in its own thread, responding to its own signals. **It is therefore important that you define a different set of handlers for each task type in your program**.

Task Accept signal list

Task handlers can specify bounded entire signals. As you know, when a signal of this kind is generated in a process, all threads of the process that have registered for the signal will in fact catch that signal. In the case of a **task** handler, the handler will be invoked as soon as the signal arrives. However, the **task** must register its acceptance of these signals. It does that by specifying **accepts**(…) list in its definition, say near its **invariant** specification or somewhere visible for readability. The specification is similar to that of **throws**(…), as shown below.

```
accepts(_sig1,  …, _sigN);
```

Task Hear Signals

The registration of hear signals is done similar to accept signals discussed in previous section, using the specification **hears**(…). In fact, receiving a hear signal is also done the same way, except hear signals may also receive data associated with the signal.

Below is an example illustrating the definition of a handler for a hear signal.

```
void MyHearHandler(void)<sig : object_1, object_2, …>;
```

The list object_1, object_2 etc. if present must be members of the **task**. The colon is only needed in presence of a data list.

The semantics is that, upon arrival of the signal, the handler `MyHearHandler` will be invoked, and the data arrived from a tell statement will be copied to the list of objects prior to the execution of user statements in the handler. The signal, once delivered (the handler was invoked) will be removed from the queue of the process.

Keep in mind that each handler can only take one single signal, of any kind. Thus, for each signal there must be a handler or the compiler will complain.

Chapter X. Global Threads

Global threads, as opposed to tasks, are indispensable. In some cases, such as applications of client-server models global threads may even provide 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 crated and the function is invoked in this new thread. **The thread that made the call is not blocked**.

Section X.1 Signaling and Events

An event is usually thought of resulting from an action external to the system such as a mouse-click. In Z++ **an event is a signal sent to a specific recipient**. In GUI illustrations we saw how events are sent to a **canvas**. The same **operator <-** is used for signaling. However, since **a signal does not have a target recipient**, Z++ uses the built-in object **signal** representing the Z47 Processor.

For illustration, let us define a signal. The base type for Z++ signals is signalEventType defined in Z++ system header file, exception.h. We define new signals by extending the base type with new signals. Below we are defining a new set of signals MyOwnSignals, which contains one new signal _SIGNAL_TerminateYourself.

```
enum MyOwnSignals : signalEventType {
      _SIGNAL_TerminateYourself
};
```

The following statement generates this signal.

```
signal <- _SIGNAL_TerminateYourself;
```

The signal is sent to the Z47 Processor, and not to a particular recipient. Any thread, can catch this signal using the following **boolean** expression.

```
signal ? _SIGNAL_TerminateYourself
```

The semantics of this expression is as follows. The question **? operator** checks system signals currently in the queue for a match. If Z47 Processor finds one, the result of the expression will be true, and Z47 Processor will remove the signal from the system queue. Otherwise, the expression will evaluate to false.

Usually this expression is used in a wait-loop as shown below.

```
do enddo(signal ? _SIGNAL_TerminateYourself);
```

Thus, one thread sends the signal to the Z47 Processor, and some other thread catches that signal.

Section X.2 Inter-Thread Communication

Signaling is an effective asynchronous mechanism for inter-thread communication. For illustration, suppose we define the following signals.

```
enum MyOwnSignals : signalEventType {
      _SIGNAL_One,
      _SIGNAL_Two,
      _SIGNAL_Three,
      _SIGNAL_Terminate
};
```

The following code segment illustrates how a thread can respond to various signals generated by other threads.

```
for (;;)

      if (signal ? _SIGNAL_Terminate) break; endif;

      if (signal ? _SIGNAL_One)
            // Perform actions for signal One
      endif;

      if (signal ? _SIGNAL_Two)
            // Perform actions for signal Two
      endif;

      if (signal ? _SIGNAL_Three)
            // Perform actions for signal Three
      endif;

endfor;
```

Chapter XI. Collections

Type constructor **collection** was introduced in Section I.20. In this chapter we will illustrate the use of **shared** methods, and derivation.

We use the same example in Section I.20 with some modifications, as shown below. As you recall, we need an enumeration, and a set of **class** types for literal values of a **collection**.

Below is the enumeration for defining the **collection**.

```
enum basicFigures {_square, _rectangle, _triangle};
```

The values of **collection** will be derived from Shape. However, as far as **collection** is concerned, the classes used for its values do not have to be related in any way.

```
class Shape
    //private members, methods
public:
    void PrintArea(void);
end;

class Square : Shape  // First value
    //private members, methods
public:
    void PrintArea(void);
end;

class Rectangle : Shape  // Second value
    //private members, methods
public:
    void PrintArea(void);
end;

class Triangle : Shape  // Third value
    //private members, methods
public:
    void PrintArea(void);
end;
```

Here is the **collection** that we will use for illustration.

```
collection basicFiguresType<basicFigures> {
            _square<Square>,
            _rectangle<Rectangle>,
            _triangle<Triangle>

shared:
    void PrintArea(void);
};
```

Section. XI.1 Derivation (inheritance)

Derivation mechanism for **collection** types is single inheritance. In order to derive a new **collection** type form an existing one, first you need to extend its associated **enumeration** type. Let us extend the enumeration type `enum basicFigures.`

```
enum moreFigures : basicFigures {_parallelogram, _diamond};
```

Next we will need two new values, a Parallelogram and a Diamond.

```
class Parallelogram : Shape  // first new value
    //private members, methods
public:
    void PrintArea(void);
end;

class Diamond : Shape  // Second new value
    //private members, methods
public:
    void PrintArea(void);
end;
```

Our derived **collection** takes the following form.

```
collection moreFiguresType<moreFigures> : basicFiguresType {
            _parallelogram<Parallelogram>,
            _diamond<Diamond>
};
```

Methods can be redefined, or more methods added. In particular, more **shared** methods can be introduced, but **shared** methods of base cannot be repeated. As explained in next section, they are inherited.

The new **collection** type has access to its base type as usual. For instance, if the method `Initialize()` is defined in the base type `basicFiguresType` it can be reached in methods of the new type as `basicFiguresType::Initialize();`.

Section XI.2 Shared Methods

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.

- Only methods can appear in the **shared** section.
- Access of the **shared** section will be whatever it was just before the **shared** keyword was visited, as to **private**, **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 probably the most interesting part, in particular that the compiler generates the body for a **shared** method of a **collection**. 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**.

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 on all values of a **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.

Chapter XII. Database Reference

Z++ database abstraction is intended for interaction with an existing database. Therefore, the abstraction covers the Data Manipulation Language (**DML**) of **SQL** standard.

Z++ database statements are **object-oriented** in nature, even though they resemble SQL statements. Moreover, Z++ statements handle database identifiers for tables and fields, as well as Z++ objects, seamlessly.

Database operations require establishing a session, which includes connecting to the server and logging in. One then needs to end the session. These operations have been abstracted away through declaring a database object. At elaboration a session is established, which terminates when the object goes out of scope.

The Z++ type of database object is defined in the system include file database.h. All operations are implemented in the standard static library. However, beyond a plain declaration, there is no need to be aware of the details of the methods that the database system type defines. **All methods are used internally by, the Z++ compiler**.

Z++ database statements are associated with a container template, such as a list. For instance, one executes the select statement in order to receive all records that satisfy the conditions of a query. Z++ select statement will automatically collect all such records in the template list associated with the statement. We use the term **catalyst** to refer to the type that instantiates the container template.

Section XII.1 Establishing a Session

The following lines are needed for using the database library.

```
#include<database.h>
using namespace databaseSpace;
```

In order to begin a database session, the instances of database types, `databaseType` and `databaseUserType`, must be constructed.

```
databaseType Dbase(IP-address, port, database-name);
databaseUserType Duser(Dbase, user-name, password);
```

First, object `Dbase` is constructed using the IP-address of database (DBMS) server, port number and the name of the database we wish to connect to. The session is established with the construction of the object `Duser`, which uses the object `Dbase`. The object `Duser` needs the user's name and password for logging into the database server.

The session will persist until the database objects we constructed go out of scope. **It is possible to establish multiple sessions with same or different servers, running different Database Management Systems**. The only requirement is that the management system supports the SQL standard.

The object `Dbase` takes more arguments for various purposes. We will discuss these in later sections, as well as the default values for the parameters.

Section XII.2 Preliminary Definitions

In this section we define a few objects so we can use them for illustrative purposes in the following section. We assume the database we are connecting to is called `MyDatabase`, and that the table in `MyDatabase` that we will use is called `MyTable`.

We assume the records in the table `MyTable` have two fields, `FirstName` and `LastName`. Accordingly we define the following **catalyst**.

```
struct record
      string first;
      string last;
end;
```

For our **template** container we will use the Z++ **List template**. Thus, `record` will be used as **catalyst** for instantiating the **List** template, as shown below.

```
#include<Iterator.h>
using namespace ztlListIteratorSpace;
List<record> recordList;
```

`recordList` will be the container for collecting the result of a query. We will also need an iterator, which we define below.

```
Iterator<record> ITR;
```

Section XII.3 Field Mapping

Database statements must map fields of a table to members of a class (the **catalyst**). With the definitions in the previous section, a mapping would be as follows.

```
MyTable.FirstName<first>, MyTable.LastName<last>
```

This maps the fields of a record in the table `MyTable` to the members of the **catalyst** object `record`. Note that the **catalyst** is the type instantiating **List**.

Section XII.4 Select Statement

For a select statement, we need to specify which container will collect the result of the query, and which method of that container will be used to populate the container. This is what the **`using`** statement below does.

```
databaseSelect<Duser, record : MyTable>
      MyTable.FirstName<first>,
```

```
      MyTable.LastName<last>
      using recordList, append;
```

databaseSelect is the Z++ keyword for **SQL** select statement. The first three arguments to **databaseSelect** are the database object `Duser`, the **catalyst** object `record` and following the colon is the name of the table we are using, `MyTable`.

You can use a comma-separated sequence of table names, in which case **Z47 Processor will perform a join**. The next two lines are mappings as we discussed in previous section. The statement ends with **using** phrase.

The semantics is that, the result of the query will populate the list `recordList,` using its method `append`. Furthermore, the fields of records obtained for the query will be copied to the members of the object `record` as indicated by the mapping.

The select statement can use **when**, **where** etc. as one would use in an **SQL** statement. We will see some illustrations with other SQL statements in the following sections. Conditional statements come just before the **using** phrase of the select statement.

Section XII.5 Deleting Records

In this section we use the iterator `ITR` defined in Section 2. Consider the following.

```
for (ITR = recordList; ITR; ITR++)
      if (ITR.Current().first == "Joe")
            databaseRemove<Duser, record : MyTable>
                  MyTable.FirstName<first>
                  where (FirstName == {ITR.Current().first});
      endif;
endfor;
```

First, we note that we are iterating over the elements in the container `recordList`. Inside the for-loop we are checking whether the first name of the current record that the iterator is pointing to, matches our chosen name, in this case "`Joe`". If it does match, we execute the Z++ **databaseRemove**. The first three arguments are the same as for select statement. The header phrase is, followed by a single mapping of the one field that we need, namely `FirstName`.

The last phrase of **databaseRemove** is the **where**-phrase. First note that after the mapping we can use the identifiers for fields of tables without qualification, in this case the identifier `FirstName`.

The expression `{ITR.Current().first}` is enclosed within curly braces {}. This is how Z++ expressions are included in a database statement. A single identifier, as opposed to an expression does not need the curly braces.

The where-clause will be checked by the database. If there is a match, DBMS will remove the record.

Section XII.6 Updating Records

In this section we use the iterator `ITR` defined in Section 2. Consider the following.

```
for (ITR = recordList; ITR; ITR++)
      if (ITR.Current().first == "MyName")
            databaseUpdate<Duser, record : MyTable>
                  MyTable.FirstName<first>,
                  MyTable.LastName<last>

                  set FirstName = "Jack", Lastname = "Jackson"
                  where (FirstName == {ITR.Current().first});
      endif;
endfor;
```

In the for-loop we iterate over the elements in the container `recordList`. Once we find a record that we wish to update, we execute **`databaseUpdate`**.

The phrases of the **`databaseUpdate`** statement should be familiar. The first three arguments are the same as those for the select statement. Then, mapping of table-fields to members of **catalyst** object are followed, as in select statement.

The last two phrases are familiar **SQL** phrases, **set** and **where**. In particular the **where**-phrase is the same as the one in previous section for **`databaseRemove`**.

Section XII.7 Inserting Records

In this section we use the iterator `ITR` defined in Section 2. Consider the following.

```
for (ITR = recordList; ITR; ITR++)
      databaseInsert<Duser, record : MyTable>
            MyTable.FirstName<first>,
            MyTable.LastName<last>
            using ITR.Current();
endfor;
```

This will insert the entire contents of the list `recordList` into the table `MyTable`. The **using** phrase uses the object currently pointed to by the iterator `ITR` for insertion. All other phrases where explained in previous sections.

Section XII.8 Database kind and Fetch size

The type `databaseType` was introduced in Section 1 as follows.

```
databaseType Dbase(IP-address, port, database-name);
```

The constructor takes a few more arguments. In this section we look at some of them, and in the next section we complete the list of arguments.

Database kind refers to well-known DBMS such as MySQL or Oracle. The default is MySQL. However, in order to specify that for definiteness, you supply that as an argument, as we will illustrate below.

The next argument is the **fetch** size. Ordinarily, the select statement gets the entire result of the query. But you can specify a number to fetch, and iterate fetching more until all records are received.

The parameter just before the database kind is a string that you may need to pass to DBMS for establishing a session. Usually, this parameter is not needed. Below we are using the empty string so the order of parameters will be preserved.

```
databaseType Dbase(IP-address,
                   port-number,
                   database-name,
                   "",           // server parameter if needed
                   _Database_Kind_Mysql,
                   10);          // number of records to fetch
```

When you set the size of fetch, the select statement will only fetch up to the fetch size, in our case 10 records. Therefore, after executing the select statement, you will need to execute the fetch statement, which returns a `boolean`. When fetch returns false, there are no more records to fetch.

In the following example, we first use the select statement of Section 4, and then follow that with a fetch statement in a do-loop. **Note that the fetch statement is identical to the select statement**.

```
databaseSelect<Duser, record : MyTable>
      MyTable.FirstName<first>,
      MyTable.LastName<last>
      using recordList, append;

boolean result;

do
      result = databaseFetch<Duser, record : MyTable>
            MyTable.FirstName<first>,
            MyTable.LastName<last>
            using recordList, append;
enddo(!result);
```

If inside the loop you make a logic that may exit the loop prior to retrieving all records resulting from a query, you can clear the database buffer as shown below.

```
databaseFree<Duser>;
```

This statement is harmless to execute even if there is nothing to free.

Section XII.9 Z++ Proxy Database and Mobile devices

The last argument to `databaseType`, after the fetch size, is a URL to the Z++ Database Proxy. The default is the string "local" which means **no proxy**.

The notion of proxy is useful for mobile devices such as PDA and cell-phones. There are two reasons to use the Z++ Proxy with regard to mobile devices. First, the result of query could be very large. Second, Z++ language is abstract and independent of platform. However, mobile devices do not directly support **SQL** standard.

The Z++ Proxy is a built-in feature of the Z++ Internet Server. All you need is the URL to where you are running the Server. **The Z++ Internet Server will perform all the database statements transparently, and act as a buffer for the mobile device**. This simply means that you do not need to make changes to your program when moving it from desktop to PDA, except for adding a URL for proxy parameter of `databaseType`.

Adding the final argument to the constructor of `databaseType` takes the following form.

```
databaseType Dbase(IP-address,
                   port-number,
                   database-name,
                   "",           // server parameter if needed
                   _Database_Kind_Mysql,
                   10,           // number of records to fetch
                   URL to Z++ Internet Server);
```

Section XII.10 Database Exceptions

The following exceptions could be raised when declaring a database object, i.e. an instance of `databaseUserType`.

```
_EXCEPTION_DATABASE_UnsupportedDatabaseKind
_EXCEPTION_DATABASE_LibraryInitializationFailed
_EXCEPTION_DATABASE_ConnectionToServerFailed
```

The following exceptions can occur when executing an SQL statement. The first three exceptions are related to the select statement. The fetch exception could also occur when executing an explicit fetch-statement.

```
_EXCEPTION_DATABASE_SelectQueryFailed
_EXCEPTION_DATABASE_InsufficientMemoryForQueryResult
_EXCEPTION_DATABASE_FetchFailed

_EXCEPTION_DATABASE_InsertRequestFailed
_EXCEPTION_DATABASE_UpdateRequestFailed
_EXCEPTION_DATABASE_RemoveRequestFailed
```

Chapter XIII. Debugging Facilities

The traditional (linguistic) debugging facility is usually included as an Assert statement. Z++ provides this facility but uses the keyword **break**. In addition, Z++ allows an entire section of your code to compile in debug mode only.

Failed Assertion

An assertion is a **boolean** expression. When the expression evaluates to false one would like the execution to stop at that line for debugging. The following statement does just that.

```
break(boolean_expression);
```

This statement is only compiled in debug mode. During execution (of debug executable) if the `boolean_expression` evaluates to false, execution will stop at that line as if a break point was set there.

Debug Section

A Z++ debug section is a **scope** that is only compiled in debug mode. You can create objects in the debug section as you need, and they will be destroyed as execution leaves the debug section. A debug section is signified as follows.

```
debug
  // debugging code in this scope
enddebug;
```

Chapter XIV. Atomic (Critical) Section

The purpose of an atomic section is to ensure that a thread performs all the statements in the specified section, without interruption.

When a thread enters an atomic section, no other thread is allowed to run until the thread leaves the atomic section.

An atomic section is a scope, meaning that the objects created within the block will be destroyed when the thread's execution leaves the atomic section. An atomic section is signified as follows.

```
atomic
  // body of atomic section
endatomic;
```

Mutex

Mutex is treated as a built-in type. To create a **mutex** simply use a declaration statement. Increment and decrement operators perform locking and unlocking. A **mutex** is destroyed when it goes out of scope. An illustration follows.

```
mutex m; // create a mutex in this scope
m++; // lock the mutex
m--; // unlock the mutex
```

A **mutex** must be passed by reference, or as pointer. Assignment of **mutex** objects is not allowed.

During a debug session, the value of a **mutex** is shown as locked, or unlocked. When a **mutex** is locked, its value also shows the number of threads waiting on that **mutex**.

Chapter XV. Preprocessor Commands

Z++ provides all preprocessing facilities for developing readable programs. All preprocessor commands must begin with the symbol #, as in C++, and must not be ended with a semicolon.

include

The **include** command follows the usual conventions. If name of file is enclosed between quotes, the search will be done in the current path, and paths set via the IDE. If filename is enclosed between <> system paths will be search. The IDE allows adding system paths in addition to the default compiler path.

define

In Z++ the **define** command is only for introducing an identifier as true for conditional compilation. It does not define a macro. See macro command in this Chapter.

undef

The **undef** command makes the identifier defined as true by the **define** command, to evaluate to false for conditional compilation.

conditional

The conditional commands consist of the following set, with their usual meaning as indicated by the command name. A sequence starts with either **if** or **ifnot**, it may contain any number of **elsif** and **elsifnot**, a single **else**, and must be closed with **endif**.

if, ifnot, elsif, elsifnot, else, endif.

macro

The **macro** command is for defining macros. Z++ macros allow multiple lines rather than one single long string. Furthermore, macros can include all preprocessor commands except **include** and **macro**. In particular, calls to other macros can take locally or globally defined objects, or argument passed to the including macro, for their arguments.

The definition of a macro, even without parameters requires a pair of parentheses. However, no parentheses should be used in 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
```

Important Note 1. The macro line can be broken into several lines. However, everything after the closing) will be ignored on same line, as indicated below.

```
#macro twoArguments(First,
                  Second) // everything else on this line ignored
      // start first line of code here
#endmacro
```

Thus, you should start the first line of code in the body of a macro on next line.

endmacro

This command simply marks end of a macro definition, like a function definition.

Continuation Character

The 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**.

Chapter XVI. Z++ Operators

When an operator has a clear meaning in the context in which it is used, it makes the statement more readable. Long names, sometimes only differentiated through the use of underscores or the case of some of their letters are not necessarily readable without confusing one for another.

Scope Operator

The scope operator is represented with two consecutive colons, "::". It is used in relation to namespaces, and types, as in `left_side::right_side`. When used for namespace scope resolution without left_side, the right_side it references comes from the global space.

The scope operator can be used recursively for nested namespaces, or types. With regard to types it is used for reaching bases or methods. Since enumerations in Z++ can also be extended, the scope operator is used to traverse a nested chain of enumeration types until the desired type is reached.

Structure Operators

The following operators are used in context of structure types like class, task or frame.

`.`	member/method selector
`->`	pointer member/method selector
`::`	base (and base member/method) selector
`->>`	pointer base selector

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 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.

The following operators are also used in connection with structure type definitions.

`:`	indicates the start of list of types for derivation (inheritance).

The colon operator is also used in extending enumeration and collection types.

`:=`	associates a frame to a GUI canvas
`=`	specifies the type as representing a module

Pointer Operators

Pointer arithmetic is same as in C++.

*		pointer de-reference
&		take address of object
()		pointer casting
++		increment pointer by size of object it is pointing to
--		decrement pointer by size of object it is pointing to
+	+=	add number of bytes to a pointer
-	-=	subtract number of bytes from a pointer

Arithmetic Operators

The following are arithmetic operators with their usual meaning as in C++. The addition from Z++ is the exponentiation operator.

=		assignment
+	+=	add
-	-=	subtract
*	*=	multiply
/	/=	divide
%	%=	modulus (remainder)
*^	*^=	exponent (raising to a power)
&	&=	bit-wise and
|	|=	bit-wise or
^	^=	bit-wise exclusive or
<<	<<=	shift-left
>>	>>=	shift-right

Operators << and >> are also used for input and output, as in C++. The unary operators are as follows.

+	no action
-	change sign
~	invert bits
++	increment (enumeration successor)
--	decrement (enumeration predecessor)

In Z++ increment and decrement operators can be applied to enumeration type to move to successor or predecessor literals. The bit inversion operator is also used for specification the destructor for a structure type.

Notes. Only for overloading purposes, the postscript versions of increment operator ++ and the decrement operator -- use the operator forms +@ and -@. The C++ solution is to assume an integer type argument. Note that these operators are only used in defining a class (and its methods). When applying the operators use the standard forms ++ and --.

Logical Operators

Z++ supports long evaluation of logical operators in which both operands are evaluated. In contrast, in short circuit evaluation, the operands are evaluated to the extent that it is necessary to determine the outcome. Z++ also adds an exclusive or to the list of logical operators.

`!`	negation
`&&`	short and (same as C++)
`||`	short or (same as C++)
`<>`	long and (both operands are evaluated)
`><`	long or (both operands are evaluated)
`^^`	exclusive or (both operands are evaluated)

Relational Operators

`==`	equal
`!=`	unequal
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal

Z++ allows a short hand form for chaining these operators. For instance, instead of the following expression,

```
(a < b && b <= c && c == d)
```

one can write `(a < b <= c == d)`. The latter looks more mathematical and compact.

Signaling Operators

`<-`	send a signal
`?`	receive a signal

Chapter XVII. Autonomous Agents

An autonomous agent is a Z++ application that moves from one node to another while running as a process of Z47 processor. That means, once the process reaches its destination it will continue to run as if it never moved. Specifically, that means that an **autonomous agent is oblivious of its teleportation**.

The mechanism for moving an autonomous agent is also invisible to the engineer. In Z++ language, there is one simple statement that does all the work.

```
travel destination_url;
```

The term **travel** is a Z++ keyword, and the destination URL is an IP-address to go to, of type **string**.

It may be worthwhile to emphasize that a Z++ autonomous agent is a complete application. This is in contrast to sending instances of classes to a destination. In the latter case, most probably one also has to code such things as proxy and adaptor classes. As mentioned earlier, **in Z++ the process of teleportation is transparent to Z++ engineers**, which implies that there will be no testing or maintenance related to code for transporting the agent.

At an airport, an arriving plane can receive an autonomous agent from the control tower that could automate all its operations until landing. In fact, if there is a solution for such automation it can only be done with the technology of autonomous agents. An autonomous agent can also act like a secretary on a smart phone (PDA). The agent can travel to sites, make reservations and inform the owner of the confirmation, or none availability of the requested service.

Chapter XVIII. Communicating Concurrent Processes

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.

Z++ presents CCP via tell-hear signaling. In time we will add more illustrative examples to this section.

Chapter XIX. Asynchronous Function Call

The tell-hear mechanism 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. However, 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 opaque to the engineer writing a program that needs to look for certain services on the web.