

[Introduction : Structure of a Z++ Program](#)
[Lesson 1 : Fundamental Types](#)
[Lesson 2 : Control Structures \(Selection\)](#)
[Lesson 3 : Control Structures \(Iteration\)](#)
[Lesson 4 : Unions, Structures and Classes](#)
[Lesson 5 : Templates and Casting](#)
[Lesson 6 : Threads](#)
[Lesson 7 : Modules and Components](#)
[Lesson 8 : Exceptions](#)
[Lesson 9 : Revisiting Enumeration](#)
[Lesson 10 : Conditional and Comparison Operators](#)
[Lesson 11 : Arrays \(operator overloading\)](#)
[Lesson 12 : Dynamic Arrays](#)
[Lesson 13 : Degenerate Array Pointers](#)
[Lesson 14 : Graphical User Interface](#)
[Lesson 15 : Frames](#)
[Lesson 16 : Platform-free GUI operations](#)
[Lesson 17 : Message Boxes](#)
[Lesson 18 : Global Threads](#)
[Lesson 19 : Common \(static\) Members](#)
[Lesson 20 : Namespaces](#)
[Lesson 21 : Class Invariants](#)
[Lesson 22 : Method Constraints](#)
[Lesson 23 : Collections](#)
[Lesson24: Travel \(Mobile Agent\)](#)

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 is that simple.

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 toil-free, simple 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. Transmissions are in small binary packets, and without the need to convert them back and forth from text.

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 the lesson 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.

```
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.

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".

The values of an enum type, like "_apple", are not global, as is the case with C++. 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 a future lesson.

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 enum instances, 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 (as in C++), or compared to integers. We shall see how the bracket function retrieves integer values associated with enumeration literals in a future lesson.

Control Structures: Selection

In this lesson we illustrate if ... else, conditional and switch statements.

Remark. 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.

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, in a future lesson 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
    //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". 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, as it happens in C++ (even when you simply forgot to put a break).

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 with at least 10 errors. 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 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 a literal value of an object. It can simply be used the same way 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
```

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

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** because all redefined methods in a derivation are made virtual by the compiler.
2. Instead of C++ pointer **this**, Z++ uses Smalltalk keyword **self**. However, self is a reference to object as opposed to pointer.
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.

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 lesson), 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. 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 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 a 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 "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 or backwards.

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 toil-free 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.

Threads

Multi-threaded programs in Z++ are toil-free and safe. The 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.

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. **Nothing more is needed to create complex multi-threaded programs that look so simple and comprehensible.**

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.

Modules and Components

A Z++ executable is also a module, a DLL, a package, a reusable component and so on, without having to do anything beyond writing your program. Furthermore, the Z++ compiler can also create static libraries.

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 component-oriented development.

The set of entry points of a module constitutes its boundary. A module is invoked by indicating an entry point, and passing the arguments expected by the selected entry point.

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. 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 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.** That is, when the object is created the state of its module is initialized. 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, by calling their destructors.

Remark. 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.

Remark. The Z47 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 the remote machine may execute the code of the module on behalf of the local machine.

Exceptions

Z++ slightly 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 file `exception.h`. To define your own exceptions just extend the set as shown in Lesson One.

The linguistic construct takes the following syntax.

```
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 word "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 will service 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 level without being serviced, your program will be terminated.

Here is how you raise your own exceptions.

```
raise some_exception;
```

Remark. 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 an introductory lesson, 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 in Lesson Two, 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.

Revisiting Enumeration

Enumeration was introduced in Lesson One. 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;
```

Thus, [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.

Conditional and Comparison Operators

The conditional 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. The operator || is treated analogously.

In Z++ one can force the evaluation of operands. The full-circuit operator for and is <> and the one for or is ><. Thus, 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 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)
```

Arrays: operator overloading

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

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

//code to set values for cells of a

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

//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.

C++ 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 a, b and c. The constructor is applied to every cell of array. This is not possible in C++.

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 (second operand for `size`).

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
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.

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];
```

Graphical User Interface

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

The screen used by tools to create GUI is called a canvas. Once the canvas is prepared, the tool generates a small 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, as you must in any other language. 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 lesson.

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.

Frames

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 only 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,
    _IES_Pen_Hold_Signal,
    _IES_Pen_Drag_Signal,
    _IES_Pen_Drop_Signal,
    _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 pen 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. 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 body of an instinct method. For now 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_Pen_Tap_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 pen, you use select on the entity member of the argument. It is that simple.

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 method and pass the argument to it. You simply focus on your logic.

Platform-free 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 lesson 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 (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 check mark for One

    return;

case _IES_Erase_Signal:
    $self; //erase can only appear in instinct
    return;

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. It is that simple.

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. There is nothing else you need to do. 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 Operations

In this section we list several operators for reference. The preceding example illustrates their usage.

`operator <<`

This operator has been conveniently overloaded with rather obvious semantics.

For certain GUI elements, only certain types are meaningful, which the compiler will enforce. For instance, one can only output a boolean to a checkbox. If the value is true, the box will be checked, otherwise the box will be unchecked.

operator >>

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

operator \$

This is the draw/erase toggle operator. We have already seen its use for drawing and 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 radiobutton called "First".

operator ~

Currently the clearing operator is applicable to fields only. It erases the contents of a field, as opposed to operator \$ which toggles the visibility of an entity. Note that clearing a field is not a toggling action. Once cleared, the contents are lost.

operator <-

In the context of GUI operations, the signal operator is used to send events to a canvas. This was illustrated by sending erase signal, in the above example code.

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 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". That is all for the buttons.

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 an error box.

```
MyCanvas << "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.

Global Threads

Object-oriented threading in Z++ is abstracted in the form of task type constructor, presented in lesson 6. 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.

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 the processor. The operator `<-` is the same one used in GUI signaling 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 it finds one, the result of the expression will be true, and the signal is removed from 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.

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, 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.

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 introduction of a namespace (via directive) 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` 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 implementation can be done in the same file, or a separate file, as follows.

```
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, the following statement 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`. Furthermore, all other C++ features, like renaming a namespace, can be done via derivation, and with more control.

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.

Method Constraints

Constraints are conditions that must hold prior to the execution of the code of a method. Rudimentary forms of constraints are called 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.

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 any type. That is, the values associated with enumeration literals are of type int, while in case of collection those values are instances of class type.

In order to define a collection, we need an enumeration and the types of 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 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 illustrated in the examples.

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 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 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.

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.

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 "12.166.5.0"; // go to the IP address

handler

    case _EXCEPTION_SendError:
        // handle exception if possible
        resume;

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.

Since agents travel from node to node and generally run in the background, it is more logical to use files for IO instead of keyboard and screen.