University of Kashmir, North Campus

Course Code	Course Name	Course Instructor
MCA-101-CR	Advanced Programming Concepts in C / C++	Bilal Ahmad Dar

Contents

1. Templates

The template is one of C++'s most sophisticated and high-powered features. Using templates, it is possible to create generic functions and classes and thus provide support for **generic programming**. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures. Thus, we can use one function or class with several different types of data without having to explicitly recode specific versions for each data type.

1.1. Function Templates/Generic Functions

Function templates are special functions that can operate with generic types. Function templates could be used to create a family of functions with different argument types. In C++, a template function is created using the keyword **template**.

The general form of a template function definition is shown here:

```
    template <class type> ret-type func-name(parameter list)
{
        // body of function
}
```

Here, ***type*** is a placeholder name for a data type used by the function.
Since C++ does not recognize end-of-line as a statement terminator, the **template** clause of a generic function definition does not have to be on the same line as the function's name.
So a template can also be written as:

```
template <class type>
ret-type func-name(parameter list)
{
        // body of function
}
```

1.1.1. Function with generic type

The following example creates a generic function that swaps the values of the two variables.

```
#include <iostream>
template <class X> void swap(X &a, X &b)            // This is a function template.
{
        X temp;
        temp = a;
        a = b;
        b = temp;
}
int main()
{
    int i=10, j=20;
    double x=10.1, y=23.3;
    char a='x', b='z';
    cout << "Original i, j: " << i << ' ' << j << '\n';
    cout << "Original x, y: " << x << ' ' << y << '\n';
    cout << "Original a, b: " << a << ' ' << b << '\n';
    swap(i, j);                 // swap integers
    swap(x, y);                 // swap floats
    swap(a, b);                 // swap chars
    cout << "Swapped i, j: " << i << ' ' << j << '\n';
    cout << "Swapped x, y: " << x << ' ' << y << '\n';
    cout << "Swapped a, b: " << a << ' ' << b << '\n';
    return 0;
}
```

```
template <class X> void swap(X &a, X &b)
```

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, **X** is a generic type that is used as a placeholder. After the **template** portion, the function **swap()** is declared, using **X** as the data type of the values that will be swapped. In **main()** , the **swap()** function is called using three different types of data: **int**s, **floats**, and **char**s. Because **swap()** is a generic function, the compiler automatically creates three versions of **swap()** : one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.

Here are some important terms related to templates. First, a generic function (that is, a function definition preceded by a **template** statement) is also called a

template function. When the compiler creates a specific version of this function, it is said to have created a ***specialization****.* This is also called a ***generated function***. The act of generating a function is referred to as ***instantiating*** it. A generated function is a specific instance of a template function.

1.1.2. A Function with Two Generic Types

We can define more than one generic data type in the **template** statement by using a comma-separated list. For example, this program creates a template function that has two generic types.

```
#include <iostream>
template <class type1, class type2>
void func(type1 x, type2 y)
{
        cout << x << ' ' << y << '\n';
}
int main()
{
        func(10, "Hello");
        func(98.6, 19);
        return 0;
}
```

In this example, the placeholder types **type1** and **type2** are replaced by the compiler with the data types **int** and **char ***, and **float** and **int**, respectively, when the compiler generates the specific instances of **func()** within **main().**

1.1.3. Explicitly Overloading a Generic Function

Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called ***explicit specialization***. If you overload a generic function, that overloaded function overrides (or "hides") the generic function relative to that specific version. For example, consider the following revised version of the argument swapping example shown earlier.

```
// Overriding a template function.
#include <iostream>
template <class X>
 void swap(X &a, X &b)
```

```
{
        X temp;
        temp = a;
        a = b;
        b = temp;
        cout << "Inside template swap.\n";
}
// This overrides the generic version of swap() for ints.
void swap(int &a, int &b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
        cout << "Inside swap int specialization.\n";
}
int main()
{
        int i=10, j=20;
        double x=10.1, y=23.3;
        char a='x', b='z';
        cout << "Original i, j: " << i << ' ' << j << '\n';
        cout << "Original x, y: " << x << ' ' << y << '\n';
        cout << "Original a, b: " << a << ' ' << b << '\n';
        swap(i, j);                     // calls explicitly overloaded swap()
        swap(x, y);                     // calls generic swap()
        swap(a, b);                     // calls generic swap()
        cout << "Swapped i, j: " << i << ' ' << j << '\n';
        cout << "Swapped x, y: " << x << ' ' << y << '\n';
        cout << "Swapped a, b: " << a << ' ' << b << '\n';
        return 0;
}
```

This program displays the following output.

```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swap int specialization.
Inside template swap.
Inside template swap.
```

```
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

1.1.4. Overloading a Function Template

In addition to creating explicit, overloaded versions of a generic function, you can also overload the **template** specification itself. To do so, simply create another version of the template that differs from any others in its parameter list. For example:

```
#include <iostream>
// First version of fun() template.
template <class X> void fun(X a)
{
        cout << "Inside fun(X a)\n";
}
// Second version of fun() template.
template <class X, class Y> void fun(X a, Y b)
{
        cout << "Inside fun(X a, Y b)\n";
}
int main()
{
        fun(10);                    // calls fun(X)
        fun(10, 20);                // calls fun(X, Y)
        return 0;
}
```

Here, the template for **fun()** is overloaded to accept either one or two parameters.

1.2. Generic Classes

In addition to generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized. For example, the same algorithms that maintain a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto part information. When you create a generic class, it can perform the operation you define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.

The general form of a generic class declaration is shown here:

template <class ***type***> class *class-name* {

..

..

}

Here, ***type*** is the placeholder type name, which will be specified when a class is instantiated. If necessary, you can define more than one generic data type using a comma-separated list.

Once you have created a generic class, you create a specific instance of that class using the following general form:

class-name <***type***> *ob*;

Here, *type* is the type name of the data that the class will be operating upon. You could also define generic variables in the class as:

type var-name;

1.2.1. Member Function Template

When defining a function as a member of a templated class, it is necessary to define it as a template function:

template <class ***type***> ret-type *class-name<**type**>::fun-name()*
{
..
..
}

Member functions of a generic class are themselves automatically generic. You need not use **template** to explicitly specify them as such.

1.2.2. Class Template example

In the following program, the **Calculator** class is created as a generic class. Thus, it can be used to calculate (Addition and multiplication) objects of any type.

```
#include <iostream>
template <class A> class Calculator                    //Template class
{
  public:
    A multiply(A x, A  y);
    A add(A x, A  y);
};
template <class A> A Calculator <A>::multiply(A x, A y)       //Member function template for
multiplication
{
  return x*y;
}
template <class A> A Calculator <A>::add(A x, A y) //Member function template for Addition
{
  return x+y;
}

int main()
```

```
{
        Calculator <int> s;                    // create integer type
        cout << "Addition of two integer values: " << s.add(10,20)<< "\n";
        cout << "Multiplication of two integer values: " <<s .multiply(10,20)<<"\n";

        Calculator<double> p;                  // create Float type
        cout << "Addition of two float values: " << p.add(5.3,7.6)<< "\n";
        cout << "Multiplication of two float values: " <<p.multiply(5.3,7.6)<<"\n";
        return 0;
}
```

Notice how the desired data type is passed inside the angle brackets. By changing the type of data specified when **Calculator** objects are created, you can change the type of data on which the calculations are to be performed.

2. Exception Handling

Exceptions are errors that occur at runtime. They are caused by a wide variety of exceptional circumstance, such as running out of memory, not being able to open a file, trying to initialize an object to an impossible value, or using an out-of-bounds index to an array. When such exceptions occur, the programmer has to decide a strategy according to which he would handle the exceptions.

2.1. Basics of Exception Handling

C++ provides a systematic, object-oriented approach to handle run-time errors. The exception mechanism of C++ uses three keywords: **try**, **catch** and **throw.** In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed.

The general form of **try** and **catch** are shown here.

```
try {

        // try block
}
catch (type1 arg) {

        // catch block

}
catch (type2 arg) {

        // catch block

}
catch (type3 arg) {

        // catch block

}
..
.
catch (typeN arg) {

        // catch block

}
```

When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement associated with a **try**. Which **catch** statement is used is determined by the type of the exception. When an exception is caught, *arg* will receive its value. Any type of data may be caught, including classes that you create. If no exception is thrown (that is, no error occurs within the **try** block), then no **catch** statement is executed.

The general form of the **throw** statement is shown here:

throw *exception*;

throw generates the exception specified by *exception*. If this exception is to be caught, then **throw** must be executed either from within a **try** block itself, or from any function called from within the **try** block (directly or indirectly). Throwing an unhandled exception causes the standard library function **terminate()** to be invoked.

2.2. Simple Example

Here is a simple example that shows the way C++ exception handling operates.

```
// A simple exception handling example.
#include <iostream>
 int main()
{
        cout << "Start\n";
        try
         {
                // start a try block
                cout << "Inside try block\n";
                throw 100;                              // throw an error
                cout << "This will not execute";
        }
        catch (int i)                           // catch an error
         {
                cout << "Caught an exception -- value is: ";
                cout << i << "\n";
        }
        cout << "End";
        return 0;
}
```

This program displays the following output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

As you can see, there is a **try** block containing three statements and a **catch(int i)** statement that processes an integer exception. Within the **try** block, only two of the three statements will execute: the first **cout** statement and the **throw**. Once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is, **catch** is *not* called. Rather, program execution is transferred to it. (The program's stack is automatically reset as needed to accomplish this.) Thus, the **cout** statement following the **throw** will never execute.

2.3. Using multiple catch Statements

As stated, you can have more than one **catch** associated with a **try**. In fact, it is common to do so. However, each **catch** must catch a different type of exception. For example, this program catches both integers and strings.

```
#include <iostream>
using namespace std;
void Xhandler(int test)
{
        try
        {
                if(test)                        //if test not equal to zero
                        throw test;
                else
                         throw "Value is zero";
        }
        catch(int i)
        {
                cout << "Caught Exception #: " << i << '\n';
        }
        catch(const char *str)
        {
                cout << "Caught a string: ";
                cout << str << '\n';
        }
}
int main()
{
        cout << "Start\n";
        Xhandler(1);
        Xhandler(2);
        Xhandler(0);
        Xhandler(3);
        cout << "End";
        return 0;
}
```

This program produces the following output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

As you can see, each **catch** statement responds only to its own type.
In general, **catch** expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other **catch** blocks are ignored.

2.4. Catching All Exceptions

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of **catch**.

```
catch(...)
{
        // process all exceptions
}
```

Here, the ellipsis matches any type of data. The following program illustrates **catch(...)**.

```
// This example catches all exceptions.
#include <iostream>
void Xhandler(int test)
{
        try
        {
                if(test==0)
                        throw test;             // throw int
                if(test==1)
                        throw 'a';          // throw char
                if(test==2)
                        throw 123.23;           // throw double
        }
        catch(...)                          // catch all exceptions
        {
                cout << "Caught One!\n";
        }
}
int main()
{
        cout << "Start\n";
        Xhandler(0);
        Xhandler(1);
        Xhandler(2);
        cout << "End";
        return 0;
}
```

This program displays the following output.

```
Start
Caught One!
Caught One!
Caught One!
End
```