GENERIC FUNCTIONS AND CLASSES IN CPP

Generic Functions

Ageneric function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.

A generic function is created using the keyword **template**. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed.

The general form of a template function definition is shown here: template <class *Ttype*> *ret-type func-name*(*parameter list*)

{

// *body of function*

}

Here, *Ttype* is a placeholder name for a data type used by the function. This name may be used within the function definition. However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function.

Although the use of the keyword **class** to specify a generic type in a **template** declaration is traditional, you may also use the keyword **typename**.

The following example creates a generic function that swaps the values of the two variables with which it is called. Because the general process of exchanging two values is independent of the type of the variables, it is a good candidate for being made into a generic function.

```
// Function template example.
#include <iostream>
using namespace std;
// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
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';
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(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;
}
```

Let's look closely at this program. The line:

```
template <class X> void swapargs(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 **swapargs( )** is declared, using **X** as the data type of the values that will be swapped. In **main( )**, the **swapargs( )** function is called using three different types of data: **int**s, **double**s, and **char**s. Because **swapargs( )** is a generic function, the compiler automatically creates three versions of **swapargs( )**: 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*. Both terms will be used interchangeably in this book. 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. Put differently, a generated function is a specific instance of a template 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.

The following example shows another common way to format the **swapargs( )** function.

```
template <class X>
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

If you use this form, it is important to understand that no other statements can occur between the **template** statement and the start of the generic function definition.

For example, the fragment shown next will not compile.

```
// This will not compile.
template <class X>
int i; // this is an error
```

```
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

As the comments imply, the **template** specification must directly precede the function definition.

A Function with Two Generic Types

You 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>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
cout << x << ' ' << y << '\n';
}
int main()
{
myfunc(10, "I like C++");
myfunc(98.6, 19L);
return 0;
}
```

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