Lecture 11: Templates

One of the contemporary sophistries of C++ programming is defining and manipulating templates. This lecture focuses on this topic. As you will soon see, templates are quite useful in providing additional flexibility to the C++ programmer. The lecture proceeds under the following captions:

- Introduction
- Creating a Template Function
- Explicitly Overloading a Template Function
- Template Classes
- Summary and Concluding Remarks

11.1 Introduction

Before proceeding with a discussion of templates, let us briefly summarize a few important concepts: You understand the importance of program variables: that a variable allocates memory space, but may contain several values at different points in time during the execution of a program; that a variable must be of a specific data type; that an external variable may be accessed from another (program) file.

You understand the significance of functions: that a function may be invoked several times in a given program; that an external function may be used by another (program) file; that overloaded functions could have different parameters and specifications, while sharing the same function name — a C++ way of implementing polymorphism; that functions may have default values.

You have been exposed to principles of inheritance and encapsulation, and how C++ classes facilitate these. Class hierarchies also facilitate reusability of code and make a programmer's job much more manageable.

You understand the principle of polymorphism — where an object, function, method, or operator may take on different forms, depending on the context. This principle is achieved in C++ through function overloading, function overriding, default arguments to functions, and operator overloading.

Any experienced software engineer will affirm that reusability of code is not an advantage, which just occurs because one is using object-oriented development tools. Rather, it is a conscious design effort. Templates also help to facilitate the creation of reusable code.

An example, consider a set of four overloaded functions to determine and return the absolute value of a number (integer, long integer, floating point or double). The functions are identical but for each function the type of the parameter is different from the other three. Figure 11-1 provides a listing of overloaded functions that may be specified for this. It is evident from this example, that having overloaded functions help, it does not always eliminate repetitive code. The only thing that is different about these four overloaded functions is the function heading. Yet, function body has to be specified four times.

What would be more effective is if we could write just one function, which accepts the type as an argument, and a second argument for the type specified. And we can: A template function allows you to write a single function that serves as an outline (a template) for a group of functions that differ only in the types of parameters they use.

Figure 11-1: Four Overloaded abs Functions

```
int abs (int x)  //for integer
 {
  if (x<0) return –x;
  else return x;
 }

long abs (long x)  //for long
{
 if (x<0) return –x;
 else return x;
}

double abs (double x)  //for double
{
 if (x<0) return –x;
 else return x;
}

float abs (float x)  //for float
{
 if (x<0) return –x;
 else return x;
}
```

11.2 Creating a Template Function

A template function is a generic function, which defines a set of operations that will be applied to various types of data. The type of the data that the function will operate on is passed to it as an argument.

To fully appreciate the power and usefulness of template functions, consider a sorting algorithm (you will study sorting in your course in Data Structures and Algorithms) – Exchange-Selection-Sort, which sorts a list of similar items in a specific order (ascending or descending). The list could be a list of integers, floating point numbers, names, letters, or more complex objects. If we have a generic **SelectionSort** function, we could use it to sort a list of objects belonging to any type, by simply specifying on the call, an argument that has the base type of the items to be sorted. The alternative to this would be to define separate (perhaps overloaded) **SelectionSort** functions for each base type of interest, but that requires much more work.

The forgoing discussion emphasizes an important concept: a template function can automatically overload itself. This is done each time the function has to work on data of a different data type.

11.2 Creating a Template Function (continued)

A template function definition includes the following:

- The keyword **template**
- A left angle brackets (<)
- A list of generic types, separated by comma(s)
- A right angle bracket (>)

Each genetic type has two parts:

- The keyboard **class** or **typename**
- An identifier that represents the generic type (the convention is to use T, but you may use more meaningful names).

Figure 11-2: Syntax for Template Definition

```
TemplateDef ::=
template "<"class | typename Ttype {, class | typename Ttype}">"
```

Be sure to observe the following guidelines:

1. The angular bracket is part of the syntax, and does not mean, programmer-supplied as in previous BNF notations, hence the use of quotation marks.

2. The keyword **class** is used by tradition. An alternate and perhaps more useful keyword is **typename**. In either case, **Ttype** is simply a placeholder for the actual data type (which may be scalar or advanced) used by the function when it is called.

3. The template definition informs the C++ compiler that what follows is a template function whose data type will be substituted when the function is called.

Having defined the template types, we can now define a template function that uses the templates as figure 11-3 illustrates.

Figure 11-3a: Illustrating Specification of a Template abs Function

```
template <typename Ttype> // template <class Ttype>

Ttype abs (Ttype x)
{
  if (x<0) return –x;
  else return x;
}
// …
int main(int argc, char *argv[])
{
  int i =10, j, k;
  float x=10, y, z;
  j = i - 20;
  y = x - 40;
  k = abs<int>(i); // k = abs(i);   // yields an integer result
  z = abs<float>(x);  // z = abs(x); // yields a floating point result
  //…
 }
```

When a template is specified, several genetic functions may use it. An alternate, but perhaps much clearer approach is to merge the first two lines of the code in figure 11-3a, as illustrated in figure 11-3b.

Figure 11-3b: Alternate Specification of a Template abs Function

```
template <typename Abstype> AbsType abs (AbsType x)
{
  if (x<0) return –x;
  else return x;
}
// … As in figure 11-3a
```

Here, it is very clear that **abs** is a template function, making use of the template type holder **AbsType**. The general format of this approach is:

```
template <typename Ttype {, typename Ttype}> ReturnType FunctionName (ParmList)
{
  // Function body
}
```

11.2 Creating a Template Function (continued)

Following are some additional insights for your attention:

1. Every time a template function is used with a different type or set of types, an instance (or specialization) of the template function is generated. This in effect, constitutes *implicit function overloading*, but requires less effort on the part of the programmer.

2. A generic (template) function can have, in addition to generic type parameters, standard parameters. These are specified in the normal manner.

3. As implied from the definition given earlier, a generic function may require more than one template data types. Figure 11-4 provides an illustration of this.

Figure 11-4: Illustrating a Template Function Using Multiple Template Data Types

```
template <typename Type1, typename Type2>
void Echo (Type1 x, Type2 y)
// This function prompts two variables of possible different types on the same line
{
  cout << “x: “ ; cin >> x; char ch = getchar();
  cout << “y: “ ; cin >> y; ch = getchar();

  cout << “x is “ << x << “ and y is “ << y << endl;
  cout << “Press any key to continue…”;
  ch = getchar();
}
```

11.3 Explicitly Overloading a Template Function

It has already been established that a template function overloads itself whenever this becomes necessary. To add more sophistication, you can explicitly overload a template function — a process described as explicit specialization. The overloaded function overrides (hides) the genetic function, relative to that specific scenario. Figure 11-5 illustrates this concept. In the figure, the explicitly overloaded **abs** function will override any instance of the template function of the same name whenever **int** is the return type.

Figure 11-5: Illustrating Explicit Overloading of a Template Function

```
template <typename AbsType >  AbsType abs (AbsType x)
{
  if (x<0) return –x;
  else return x;
}

int abs (int x)  // Explicitly overloaded version
{
  if  (x<0) return –x;
 else return x;
}
```

In addition to explicitly overloading the generic function, you can also overload the function template specification, by simply creating multiple versions that differ from each other. In overloading the function template, what you are in effect doing, is creating alternate version(s) of the generic function. Figure 11-6 illustrates this.

Figure 11-6: Illustrating Explicit Overloading of a Template Function

```
template <typename Type1>  void aFunctionA (Type1 x)
{
… // Function body
}

//  A second version of the template function that uses two arguments
template <typename Type1, typename Type2>  void aFunctionA (Type1 x, Type2 y)
{
…  // Function body
}
```

11.4 Template Classes

In addition to template (generic) functions, you can also define template (generic) classes. A template class defines all the algorithms used by that class. However, the actual type of data being manipulated is specified as a parameter when objects of the class are created.

Template classes are useful when a class uses logic that can be generalized. In your course on Data Structures and Algorithms, you will learn about various data structures such as stacks, queries, linked lists, trees and graphs. In each case, the objects stored in a data structure will depend on the application. For example, you can define a linked list of telephone numbers, student objects, inventory items, or any data type that is deemed necessary. To illustrate, in the case of a linked list, the operations to be performed on the linked list will be identical — node addition, node search, node modification, node deletion, or converting the list to an array. Irrespective of the kinds of objects in the list, this will be the case. Scenarios such as this therefore warrant the use of generic classes. The general form of a template class declaration is shown in figure 11-7.

Figure 11-7: Syntax for Creating a Template Class

```
template "<"class | typename Ttype {,<class | typename Ttype>}">"
class <ClassName>
{
 //… Data members and member functions
}
```

Note: As in the previous sections, the **Ttype** is a placement holder specified by you. Again, the angular brackets immediately following the **template** keyword are required as part of the syntax, hence the use of quotation marks. The body of the class will contain at least one data member of (each) type **Ttype**.

Except for inline functions, each member function must be specified with the template for the class as follows:

```
template <class | typename Ttype {, class|typename Ttype>}>
ReturnType  ClassName <Ttype {,Ttype2…}> :: FunctionName(Parameter(s))
{
 //… The rest of the method
}
```

When instantiating an object via the template class, you must specify actual data type(s) for the placement holder(s) enclosed in angular braces and in the order in which they were declared in the template:

```
ClassName <Type(s)> object;
```

Figure 11-8 provides a program listing that makes use of a template class called **Echo**. This class allows a user to enter numeric values and re-echo those values on the monitor. Pay special attention to line 27, where instantiation is done. Notice that the data type to be applied is specified in angular brackets.

Figure 11-8: Illustrating Specification and Use of Template Class

```
#include <cstdlib>          #include <cstdlib>          #include <iostream>
using namespace std;

template <typename Ttype>
class Echo
{
protected:
 Ttype info;

 public:
 Echo() // Inline constructor
 {info = 0;}

 void inputData() // Inline input function
 {
  cout << "Key the desired value for Info: "; cin >> info;
 char dummy = getchar();
 }

 void echoBack() // Inline EchoBack
 {
  cout << " You entered the following info: " << info << endl;
 }
};

// ***********************************************************
const string HEADING = "Demonstrating Template Class";
int main(int argc, char *argv[])
{
// Declarations
 bool exitTime = false;  char exitKey;
 Echo<int> echo1;  Echo<double> echo2; // Instantiation

 while (!exitTime) // While user wishes to continue
 {
   // Prompt the user for the number of StudentC objects and create the array
   cout << endl << HEADING << endl << endl;
   echo1.inputData();    echo2.inputData();

   cout << "\n\nHere is the output: \n\n";
   echo1.echoBack();     echo2.echoBack();

   // Check whether user wishes to continue
   cout << "\n Press any key to continue or X to exit ";
   exitKey = getchar();    if (toupper(exitKey) == 'X') exitTime = true;
 }

system("PAUSE");  return EXIT_SUCCESS;
}
```

11.4.1 Template Classes with Default Arguments

You can specify a default value for a template class, in a similar manner to how you specify a default value for a function parameter. The difference is that the default value for the template class must (obviously) be a valid (scalar or programmer-defined) data type. The general syntax of the template class declaration may therefore be modified to facilitate default types, as represented in figure 11-9.

Figure 11-9: Syntax for Creating a Template Class

```
template "<"class | typename Ttype [=<Type>]  {,<class | typename Ttype> [=<Type>] }">"
class <ClassName>
{
  //… Data members and member functions
}
```

With this adjustment, line 3 of figure 11-8 could be revised to set a default data type of **long** in the following way:

```
template <typename Ttype = long> // .. No change to the rest of the code
```

If you wish to use the default data type when an instance of the class is created, simply specify an empty angular bracket. Thus, on line 27 of figure 11-8, if the default of **long** is to be used, the object instantiation statement would appear as follows:

```
Echo<> echo1;
```

11.4.3 Overloading a Template Class

As with templates functions, you can overload a template class. In so doing, you create an explicit template class specialization. Whenever the situation that warrants the use of this alternate class arises, it use will override the automatic creation of a template class instantiation by the C++ compiler. To create an explicit class specialization, be sure to use the construct **template <>** as illustrated in figure 11-10 (line 22). This alerts the C++ compiler to treat this specialization of the class as different from the other implicit specializations.

Figure 11-10: Illustrating Overload of a Template Class

```
#include <cstdlib>          #include <cstdlib>          #include <iostream>
using namespace std;

template <typename Ttype>
class Echo
{
protected:
 Ttype info;

 public:
 Echo() // Inline constructor
 {info = 0;}

 void inputData() // Inline input function
 {
  cout << "Key the desired value for Info: "; cin >> info;
 char dummy = getchar();
 }

 void echoBack() // Inline echoBack
 {
  cout << " You entered the following info: " << info << endl;
 }
};

// Overloaded Template Class
 template <>
class Echo <string>
{
  // … Specification of the overloaded template class to echo strings as well
  // …
}; // End of overloaded template class
```

11.5 Summary and Concluding Remarks

In summary, a template (generic) function allows you to specify an operation that may be applied to different kinds of data. An excellent example is writing a generic function to sort a list of items. The items may be characters, strings, numbers, or some other complex objects. The template (generic) class takes it a step further, and allows you to specify abstractions that may be applied in multiple scenarios. For example, you may develop a template class to perform various sorting algorithms, depending on the choice of the user, or a template class to manage a queue of data items – a queue of objects, strings, or numbers.

Without much thought, you should be able to appreciate that templates are very useful in providing flexibility to the programmer. When wisely used, they can significantly lessen the programming effort. To some extent, templates are similar to writing generic code in Java by making use of the **Object** class, and other generic classes that are provided by that language.

11.6 Recommended Readings

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6th Edition. Boston: Addison-Wesley. See chapter 11.

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapter 16.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapter 16.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapter 17.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapter 7.