MODULE 24
TEMPLATE

This type, that type, so many types,
No more type!
GENERIC TYPES

My Training Period: hours

Some notes:

- Well, you have completed the wave of the procedural programming then object oriented programming. In order to complete your C and C++ journey, this Module will introduce you the generic programming.
- This Module just to introduce you what the template is. The main task is to learn how to use and manipulate the STL components later.
- This Module may be very useful if you want to create our own template but at the same time it should provide us with a good understanding of the template itself.
- Compiler used in this Module is **`Visual Studio .Net® 2003`** because many of the C++ features do not supported by some of the lousy compilers included Visual Studio/C++ 6.0® with Service Pack 6 (SP6) :o) `g++` (`GNU C++`) example is given at the end of this Tutorial. With `g++`, you will be warned if there are outdated and danger constructs in your programs :o).
- The moral of this very short story is: if you want to develop programs that use many of the C++ features and don't want to get cheated by the compiler, use fully `ISO/IEC C++` compliance compiler.
- You can see that many times naïve programmers have been cheated by the compiler! You think there is something wrong with your codes, but the compiler itself does not have the capabilities to understand your codes :o) or your codes really have bugs :o).
- This Module and that follows supposed to make our tasks in programming smoother, easier, safer and more productive :o).
- Get a good understanding of the templates so that it is easier for you to use them in the next Modules.

Abilities

- Able to understand and appreciate a Template.
- Able to understand and use Function Template.
- Able to understand and use Class Template.
- Able to understand and use Template instantiation.
- Able to understand and use explicit and implicit instantiation.
- Able to understand and use Template Specialization.
- Able to understand and use Template Partial Specialization.
- Able to understand and use the keyword `typename`.

24.1 Introduction

- Many real applications use common data structure routines such as list, sort, and queue. A program may require a **List** of name and another time, a **List** of messages. So we can create a **List** of name program, and then reuse the existing code to create a **List** of messages program. Next time we may need another **List** of address etc. Again we copy the **List** of messages program. These situations happen again and again.
- If we need to change the original codes, the other codes also may need changes; at the beginning we also have to change the codes regarding the data type because **name** and **messages** may implement different data type. It will become headache isn't it?
- It is wiser to create a **List** program that contains an arbitrary data type that is ready for many data types because the **List** routine should be the same. This is called parameterized or generic data type, commonly referred to as template. Notice that the word **data type** or **type**. This is the word that we are concern about regarding why the template 'creature' exists.
- Template extends the concepts of the reusability. In the **List** example, template allows us to implement something like a generic **List** as shown below, where the `any_data_type` is its type parameter.

```
List<any_data_type>
```

- Then `any_data_type` can be replaced with actual types such as `int`, `float`, `name`, `messages`, `address` etc as shown below.

```
List<int>
List<name>
List<messages>
```

- When the changes implemented in the

```
List<any_data_type>
```

- Then it should immediately reflected in the other classes, `List<int>`, `List<name>` etc.
- Templates are very useful when implementing generic constructs such as lists, stacks, queues, vectors etc which can be used in any arbitrary data type. These generic constructs normally found in data structure, search routines and database applications.
- Designing a type-independent class enables users to choose the desired data type for specific application without having to duplicate code manually. Furthermore, type independent classes should be portable among different locales and platforms.
- It provides source code reusability whereas inheritance provides object code reusability.
- Furthermore, almost all part of the C++ Standard Library is implemented using templates.
- Generally, templates are functions or classes that are written for one or more types not yet specified. When you use a template, you pass the **types** as arguments, explicitly or implicitly.
- Basically, there are two kinds of templates:

    1. Function template.
    2. Class template.

- For example, the Standard Template Library (STL) generic **Algorithm** has been implemented using function templates whereas the **Containers** have been implemented using class template.
- We will go through the algorithm and container more detail in another Module later on.

24.2 Function Template

- Use to perform identical operations for each type of data.
- Based on the argument types provided in the calls to function, the compiler automatically **instantiates** separates object code functions to handle each type of call appropriately.
- The STL **algorithms** for example, are implemented as function templates.
- A function template declaration contains the keyword **`template`**, followed by a list of template parameters and function declaration. The definition of a function template should follow its declaration immediately as opposed to normal functions.
- For examples:

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

//function declaration and definition
template <class any_data_type>
any_data_type MyMax(any_data_type Var1, any_data_type Var2)
{
       //if var1 is bigger than Var2, then Var1 is the maximum
       return Var1 > Var2 ? Var1:Var2;
}
```

- If you have noticed, other than the red color line of code, it is same as normal function.

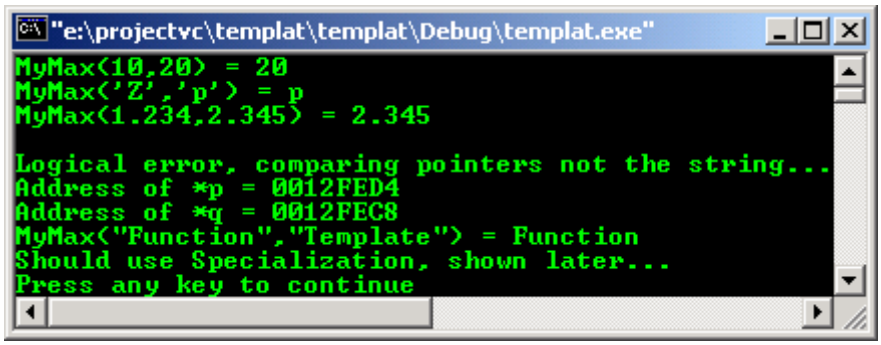
24.2.1 Function Template Instantiation

- Using function templates is similar to normal function. When the compiler sees an instantiation of the function template, for example, the call of the `MyMax(10, 20)` in function `main()`, the compiler generates a function `MyMax(int, int)`.
- Hence, it should be similar for other data type such as `MyMax(double, double)` and `MyMax(char, char)`.

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

```
//function template declaration and definition
template <class any_data_type>
any_data_type MyMax(any_data_type Var1, any_data_type Var2)
{
       return Var1> Var2 ? Var1:Var2;
}

int main()
{
      cout<<"MyMax(10,20) = "<<MyMax(10,20)<<endl;
      cout<<"MyMax('Z','p') = "<<MyMax('Z','p')<<endl;
      cout<<"MyMax(1.234,2.345) = "<<MyMax(1.234,2.345)<<endl;

      //some logical error here?
      cout<<"\nLogical error, comparing pointers not the string..."<<endl;
      char* p = "Function";
     char* q = "Template";

      cout<<"Address of *p = "<<&p<<endl;
     cout<<"Address of *q = "<<&q<<endl;
      cout<<"MyMax(\"Function\",\"Template\") = "<<MyMax(p,q)<<endl;
      cout<<"Should use Specialization, shown later..."<<endl;
      return 0;
}
```

Output:



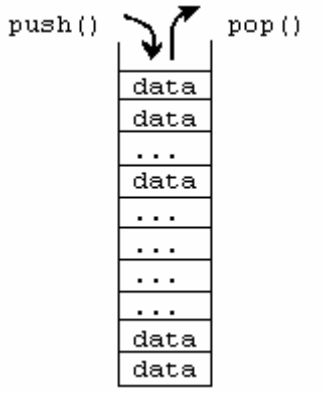
24.3 Class Template

- A class template definition look likes a regular class definition, except the keyword **`template`** and the angle brackets < >.
- It is declared using the keyword **`template`**, followed by a template parameter list enclosed in angle brackets and then a declaration and/or a definition of the class.
- For example, the following code segment is a class template definition for `MyStack`. This is not a STL `Stack` but our non standard stack template presented just for discussion.
- If you notice, other than the red color line of code, it is same as normal class, isn't it?

```
template <class any_data_type>
class MyStack
{
      private:
      //number of the stack's element
      int size;
      //top position
      int top;
      //data type pointer
      any_data_type* StackPtr;

      public:
      //constructor...
      MyStack(int =10);
      //destructor...
      ~MyStack(){delete [] StacKPtr;}
      //put in data...
      int push(const any_data_type&);
      //take out data...
      int pop(any_data_type&);
      //test the emptiness...
      int IsEmpty() const {return top == -1;}
      //test the fullness...
      int IsFull() const {return top == size - 1;}
```

```
};
```

- For your information, the top of the stack is the position occupied by the most recently added element and it should be the last element at the end of the container.
- A simple stack is illustrated below. The real story of stack construction during the function call can be read in Module W.



- Another example of the template declaration:

```
template <class any_data_type>
class Vector
{
      private:
      any_data_type *buffer;
      //copy constructor
      Vector<any_data_type>   (const Vector <any_data_type> &Var1)
      //overloaded assignment operator
      Vector<any_data_type>& operator=(const Vector<any_data_type>& Var2)
      //destructor
      ~Vector<any_data_type>();
      //other member functions…
      any_data_type& operator [ ] (unsigned int index);
      const any_data_type& operator [ ] (unsigned int index) const;
}
```

- As in class definition, member functions for class template also can be defined outside the class body. For example:

```
      //destructor definition
      template  <class any_data_type>
      MyStack<any_data_type >::~MyStack()
      {delete [ ] StackPtr;}
```

- Or

```
      //constructor definition
      template <class any_data_type>
      MyStack<any_data_type>::MyStack()
```

- `any_data_type` is data type **template parameter** and it can be any data type. It acted as a placeholder for future use; currently its types are not yet specified. For example:

```
      MyStack<MyClass>
```

- Where `MyClass` is user defined class. `any_data_type` also does not have to be a class type or a user defined type as shown in the following example.

```
      MyStack<float>
      MyStack<MessagePtr*>
```

- Or in totally generic form as shown below. Keep in mind that the `typename` is a keyword in C++ and explained at the end of this Module.

```
      template <typename any_data_type>
      class Vector
      { }
```

24.3.1 Class Template Parameters

- A template can take one or more type parameters which are the symbols that currently represent unspecified types. For example:

```
template <class any_data_type>
class Vector
{ }
```

- Here, `any_data_type` is a template parameter, also referred as type parameter.
- Another example:

```
template <class any_data_type, int p>
class Array
{ }
```

- The `any_data_type` and `p` are template parameters.
- When an ordinary type is used as parameter, the template argument must be a **constant** or a **constant expression** of an **integral type**. For example:

```
int num = 100;
const int Var1 = 10;

//should be OK, Var1 is a const
Array<float, Var1> Test;

//should be OK, 10 is a const
Array<char, 10> Test1;

//should be OK, constant expression been used
Array<unsigned char, sizeof(float)> Test2;

//Not OK, num is not a constant
Array<int, num> Test3;
```

- Besides the constant expressions, the only other arguments allowed are a pointer to a non-overloaded member, and the address of an object or a function with external linkage.
- A template can take a template as an argument. For example:

```
int ReceiveMsg(const Vector<char*>&);
int main()
{
        //a template used as an argument
        Vector <Vector<char*>  > MsgQ(20);

        //other codes….

        //receive messages
        for(int j = 0; j < 20; j++)
                ReceiveMsg(MsgQ[j]);
        return 0;
}
```

- Notice the space between the right two angle brackets, it should be mandatory as shown below to avoid the misinterpret of the right shift operator >>.

```
Vector <Vector<char*>  > MsgQ(20);
```

24.3.2 Default Type Arguments

- Class template can have default type argument same as normal class. It provides flexibility for programmer to suit her/his needs. For example the STL Vector class template, the default type `size_t` is used but the programmer is free to choose other suitable data types instead.

```
template <class any_data_type, class S = size_t>
class Vector
{ };
//second argument default to size_t
Vector <int>  TestVar;
```

```
Vector <int, unsigned char> short(7);
```

- Another example of the default template argument:

```
template <class any_data_type = float, int element = 10>
MyStack{};
```

- Hence, the following declaration:

```
MyStack<>  Var1;
```

- Would instantiate at compile time a `10` element `MyStack` template class named `Var1` of type `float`. This template class would be of type:

```
MyStack<float, 10>
```

- For template **specialization** (will be discussed later), default arguments cannot be specified in a declaration or definition. For example:

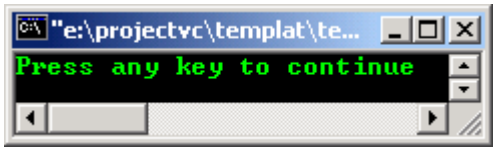
```
#include <iostream>
using namespace std;

//primary template with default parameter
template <class any_data_type, int size>
class MyStack
{};

//specialization declaration and definition
//with default arguments
//will generate error...
template <class any_data_type, int size=100>
class MyStack<int,100>
{};

//do some testing
int main()
{
       MyStack<float,100> Var1;
       return 0;
}
```

- Let try a simple template program skeleton.

```
//simple class template program example
//------declaration and definition part--------
template <class any_data_type>
class MyStack
{
     private:
     //number of the stack's element
     int size;
     //top position
     int top;
     //data type pointer
     any_data_type* StackPtr;

     public:
     //constructor...
     MyStack(int =10);
     //destructor...
     ~MyStack(){delete [] StacKPtr;}
     //put in data...
     int push(const any_data_type&);
     //take out data...
     int pop(any_data_type&);
     //test the emptiness...
     int IsEmpty() const {return top == -1;}
     //test the fullness...
     int IsFull() const {return top == size - 1;}

};

//----the main() program--------
int main()
{
```

```
        return 0;
}
```

Output:



24.3.3 Member Function Templates

- Member functions of non template classes may be templates. However, member templates cannot be `virtual`, nor may they have default parameters. For example:

```
//normal class
class MyClass
{
//...
      //but, have template member function...
      template <class any_data_type>
      void MemberFunct(any_data_type)
      {};
};

int main()
{
      return 0;
}
```

- Here, `MyClass::MemberFunct()` declares a set of member functions for parameters of any type. You can pass any argument as long as its type provides all operations used by `MemberFunct()`.

24.3.4 Nested Template Classes

- Nested classes may also be templates as shown below:

```
template <class any_data_type>
class MyClass
{
//...
      //nested class template
      template <class another_data_type>
      class NestedClass
      {};
//...
};

int main()
{ return 0; }
```

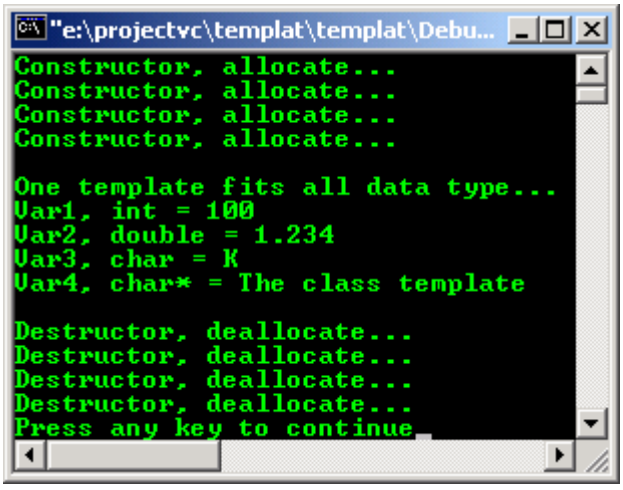
24.4 Class Template Instantiation

- A **template instantiation** is a process of instantiate a real class from a template for our real usage. It provides general class template with potentially infinite data types.
- The following is a program example of a class template instantiation. It is bundled in one program, one file for our study convenience.

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

//-------class template declaration part---
//-------test.h-----
template <class any_data_type>
class Test
{
      public:
      //constructor
      Test();
      //destructor
```

```
        ~Test();
        //function template
        any_data_type Data(any_data_type);
};

template <class any_data_type>
any_data_type Test<any_data_type>::Data(any_data_type Var0)
{return Var0;}

//------class template definition part--------
//----should be in the same header file with--
//----the class template declaration------
//constructor
template <class any_data_type>
Test<any_data_type>::Test()
{cout<<"Constructor, allocate..."<<endl;}

//destructor
template <class any_data_type>
Test<any_data_type>::~Test()
{cout<<"Destructor, deallocate..."<<endl;}

//--------main program--------
int main()
{
        Test<int> Var1;
        Test<double> Var2;
        Test<char> Var3;
        Test<char*> Var4;

        cout<<"\nOne template fits all data type..."<<endl;
        cout<<"Var1, int = "<<Var1.Data(100)<<endl;
        cout<<"Var2, double = "<<Var2.Data(1.234)<<endl;
        cout<<"Var3, char = "<<Var3.Data('K')<<endl;
        cout<<"Var4, char* = "<<Var4.Data("The class template")<<"\n\n";
        return 0;
}
```

Output:



- When repackaging the template class, the implementation of the class template is slightly different from the normal class. As mentioned before, the declaration and definition part of the class template **member functions** should all be in the same header file.
- For the previous program example, the repackaging is shown below.

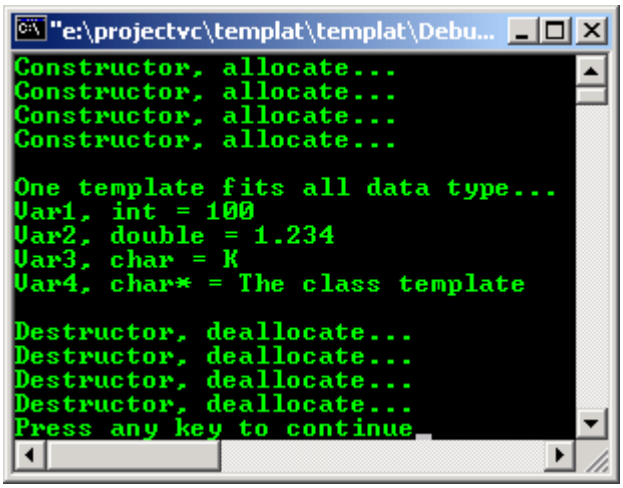
```
#include <iostream>
using namespace std;

//-------class template declaration part---
//-------test.h file-----
template <class any_data_type>
class Test
{
     public:
     //constructor
```

```
        Test();
        //destructor
        ~Test();
        //function template
        any_data_type Data(any_data_type);
};

template <class any_data_type>
any_data_type Test<any_data_type>::Data(any_data_type Var0)
{return Var0;}

//------class template definition part--------
//----should be in the same header file with--
//----the class template declaration------
//constructor
template <class any_data_type>
Test<any_data_type>::Test()
{cout<<"Constructor, allocate..."<<endl;}

//destructor
template <class any_data_type>
Test<any_data_type>::~Test()
{cout<<"Destructor, deallocate..."<<endl;}
//do not run this program
//make sure there is no error such as typo etc
```

- And the `main()` program is shown below.

```
//----test.cpp file------------
//---compile and run this program----
//--------main program--------
int main()
{
        Test<int> Var1;
        Test<double> Var2;
        Test<char> Var3;
        Test<char*> Var4;

        cout<<"\nOne template fits all data type..."<<endl;
        cout<<"Var1, int = "<<Var1.Data(100)<<endl;
        cout<<"Var2, double = "<<Var2.Data(1.234)<<endl;
        cout<<"Var3, char = "<<Var3.Data('K')<<endl;
        cout<<"Var4, char* = "<<Var4.Data("The class template")<<"\n\n";
        return 0;
}
```

Output:

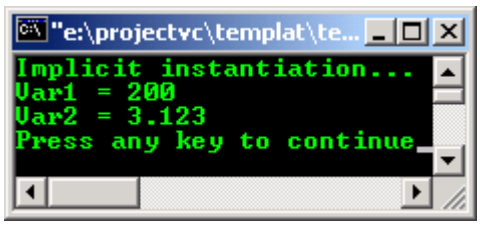


- While implementing a class template member functions, the definitions are prefixed by the keyword `template < >`.
- The compiler generates a class, function or static data members from a template when it sees an implicit instantiation or an explicit instantiation of the template. The following program example is an implicit instantiation of a class template.

```
#include <iostream>
```

```
using namespace std;

template <class any_data_type>
class Test
{
public:
     //constructor
     Test(){};
     //destructor
     ~Test(){};
     //member function templates...
     any_data_type Funct1(any_data_type Var1)
     {return Var1;}
     any_data_type Funct2(any_data_type Var2)
     {return Var2;}
};

//do some testing
int main()
{
      //Implicit instantiation generates class Test<int>...
      Test<int> Var1;
      //Implicit instantiation generates class Test<double>...
      Test<double> Var2;

      cout<<"Implicit instantiation..."<<endl;
      //and generates function Test<int>::Funct1()
      cout<<"Var1 = "<<Var1.Funct1(200)<<endl;
      //and generates function Test<double>::Funct2()
      cout<<"Var2 = "<<Var2.Funct2(3.123)<<endl;
      return 0;
}
```

Output:

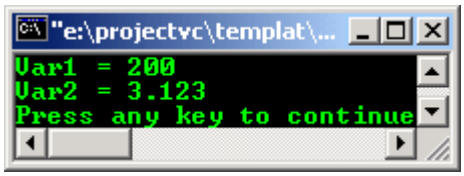


- From the program example, the compiler generates `Test<int>` and `Test<double>` classes and `Test<int>::Funct1()` and `Test<double>::Funct2()` function definitions.
- The compiler does not generate definitions for functions, non virtual member functions, class or member class that does not require instantiation.
- In the program example, the compiler did not generate any definition for `Test<int>::Funct2()` and `Test<double>::Funct1()`, since they were not required.
- The following is a program example of an explicit instantiation of a class template.

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

template <class any_data_type>
class Test
{
       public:
       //constructor
       Test(){};
       //destructor
       ~Test(){};
       //member functions...
       any_data_type Funct1(any_data_type Var1)
       {return Var1;}
       any_data_type Funct2(any_data_type Var2)
       {return Var2;}
};

//explicit instantiation of class Test<int>
template class Test<int>;
//explicit instantiation of class Test<double>
template class Test<double>;

//do some testing
```

```
int main()
{
	Test<int> Var1;
	Test<double> Var2;
	cout<<"Var1 = "<<Var1.Funct1(200)<<endl;
	cout<<"Var2 = "<<Var2.Funct2(3.123)<<endl;
	return 0;
}
```

Output:

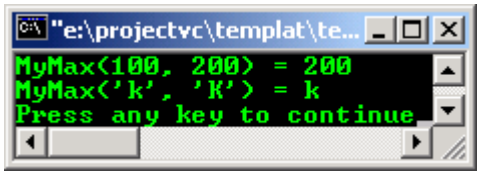


24.5 Function Template Instantiation

- The following program examples are implicit and explicit instantiation of function templates respectively.

```
//implicit instantiation
#include <iostream>
using namespace std;

template <class any_data_type>
any_data_type MyMax(any_data_type Var1, any_data_type Var2)
{
	return Var1 > Var2 ? Var1:Var2;
}

//do some testing
int main()
{
	int p;
	char q;
	p = MyMax(100, 200);
	q = MyMax('k', 'K');

	//implicit instantiation of MyMax(int, int)
	cout<<"MyMax(100, 200) = "<<p<<endl;
	//implicit instantiation of MyMax(char, char)
	cout<<"MyMax('k', 'K') = "<<q<<endl;
	return 0;
}
```

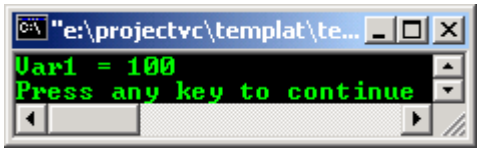
Output:




```
//explicit instantiation
#include <iostream>
using namespace std;

template <class any_data_type>
any_data_type Test(any_data_type Var1)
{
	return Var1;
}

//explicit instantiation of Test(int)
template int Test<int>(int);

//do some testing
int main()
{
	cout<<"Var1 = "<<Test(100)<<endl;
	return 0;
```

```
}
```

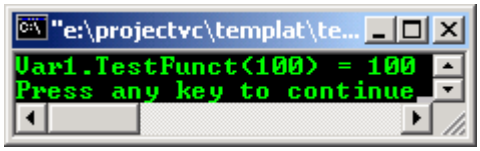
Output:



- Instantiating the virtual member functions of a class template that does not require instantiation is implementation defined.
- For example, in the following example, virtual function `TestVirt<any_data_type>::Test()` is not required, compiler will generate a definition for `TestVirt<any_data_type>::Test()`.

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

template <class any_data_type>
class TestVirt
{
        public:
       virtual any_data_type TestFunct(any_data_type Var1)
       {return Var1;}
};

//do some testing
int main()
{
        //implicit instantiation of TestVirt<int>
        TestVirt<int> Var1;

        cout<<"Var1.TestFunct(100) = "<<Var1.TestFunct(100)<<endl;
        return 0;
}
```

Output:



24.6 Class Template Specialization

- A **specialization** consists of a template name followed by a list or arguments in angle brackets and it is a specialize class template instantiation.
- When we instantiate the class template, from a general class template, we make the class template special to suit our specific programming tasks.
- A specialization can be used exactly like any other normal class and here, compiler will generate a specialized concrete instance from the generic of the templates. For examples:

```
//an object instantiation
Vector<int> Var;

//as function parameter
int Funct(Vector <float>&);

//used in sizeof expression
size_t p = sizeof(Vector <char>);

//used in class object instantiations
class MyTestVector: private Vector<std::string>
{ };
Vector <Date> Var1;
Vector <string> Var2;
```

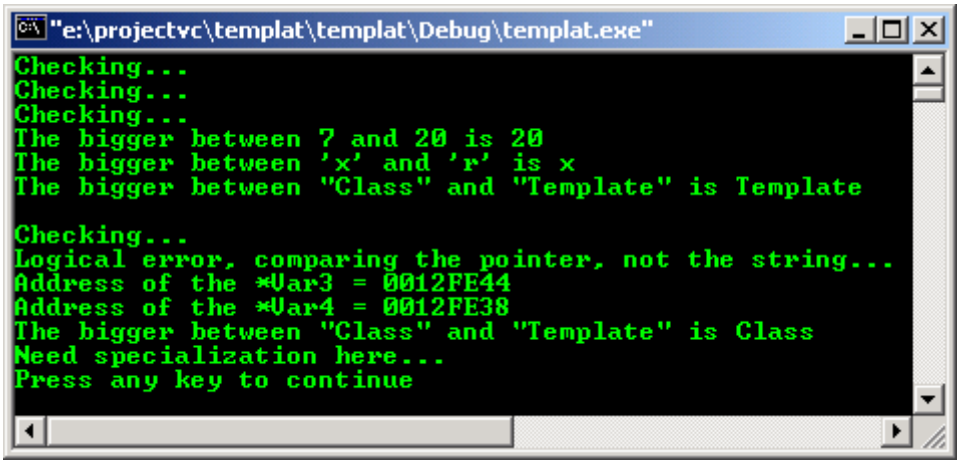
- The compiler actually instantiates only the necessary member functions of a given specialization or generate when there is a request.

24.7 The Primary, Partial and Specialization Class Template

- For example again, let take a look at the `max()` function of the STL by recreating our own version and name it `MyMax()`.
- This general class template is called **Primary Template**. It should suit to all data types. For example:

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

template <class any_data_type>
inline any_data_type MyMax(const any_data_type& Var1, const any_data_type& Var2)
{
	cout<<"Checking..."<<endl;
	return Var1 < Var2 ? Var2 : Var1;
}

//do some testing
int main()
{
	int Highest = MyMax(7, 20);
	char p = MyMax('x' , 'r');
	string Str1 = "Class", Str2 = "Template";
	string MaxStr = MyMax(Str1, Str2);

	cout<<"The bigger between 7 and 20 is "<<Highest<<endl;
	cout<<"The bigger between 'x' and 'r' is "<<p<<endl;
	cout<<"The bigger between \""<<Str1<<"\" and \""<<Str2<<"\" is
	"<<MaxStr<<"\n\n";

	const char *Var3 = "Class";
	const char *Var4 = "Template";
	const char *q = MyMax(Var3, Var4);

	cout<<"Logical error, comparing the pointer, not the string..."<<endl;
	cout<<"Address of the *Var3 = "<<&Var3<<endl;
	cout<<"Address of the *Var4 = "<<&Var4<<endl;
	cout<<"The bigger between \""<<Var3<<"\" and \""<<Var4<<"\" is "<<q<<endl;
	cout<<"Need specialization here..."<<endl;
	return 0;
}
```

Output:



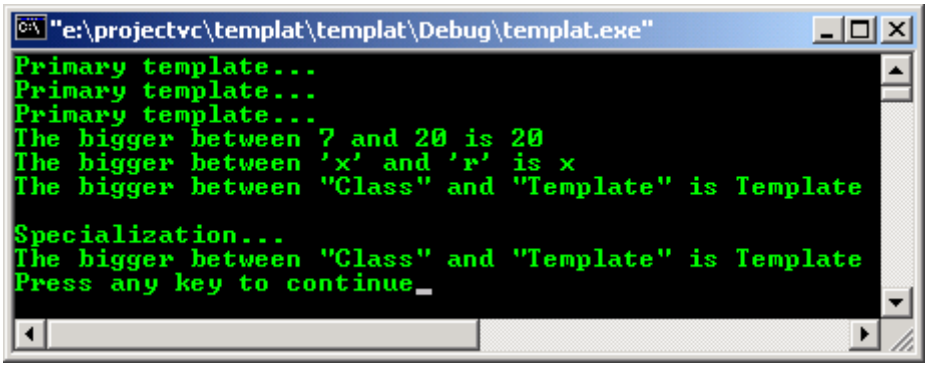
24.7.1 Implementing The Specialization

- The primary template should be generic to a potentially infinite set of template arguments. Actually, we can narrow down these generic arguments to our specific needs and refer it as user defined specialization or explicit specialization. Implicitly, instantiation should be invoked automatically for the specified data types.

- Specialization explicitly fixes all template parameters to a unique template argument. Template specialization will override the template generated code by providing special definitions for specific types.
- An **explicit specialization** looks like a normal template definition except that it must appear after its primary template.
- From the previous example, let define specialization for the `const char *` of the last part of the program example.
- Firstly we have to replace every occurrence of `any_data_type` in the specialization with `const char *`. The **template parameter list** also should be **empty** as shown below:

```
#include <iostream>
#include <string>
//for strcmp()
#include <cstring>
using namespace std;

//primary template, for all type
template <class any_data_type>
any_data_type MyMax(const any_data_type Var1, const any_data_type Var2)
{
	cout<<"Primary template..."<<endl;
	return Var1 < Var2 ? Var2 : Var1;
}

//specialization for const char *, empty parameter list
template <>
const char *MyMax(const char *Var1, const char *Var2)
{
	cout<<"Specialization..."<<endl;
	//comparison for const char *
	return (strcmp(Var1, Var2)<0) ? Var2 : Var1;
}

//do some testing
int main()
{
	 //call primary
	 int Highest = MyMax(7, 20);
	 //call primary
	 char p = MyMax('x' , 'r');
	 string Str1 = "Class", Str2 = "Template";
	 //call primary
	 string MaxStr = MyMax(Str1, Str2);

	 cout<<"The bigger between 7 and 20 is "<<Highest<<endl;
	 cout<<"The bigger between 'x' and 'r' is "<<p<<endl;
	 cout<<"The bigger between \""<<Str1<<"\" and \""<<Str2<<"\" is
	 "<<MaxStr<<"\n\n";

	 //call specialization
	 const char *Var3 = "Class";
	 const char *Var4 = "Template";
	 const char *q = MyMax(Var3, Var4);
	 cout<<"The bigger between \""<<Var3<<"\" and \""<<Var4<<"\" is "<<q<<endl;
	 return 0;
}
```

Output:

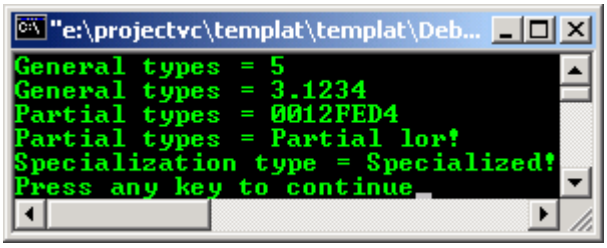


24.7.2 Partial Specialization

- Between the **primary** (general) and the **specific specialization**, is called **partial specializations**. This specialization partially fixes their template parameter which applies to a subset of types. It should suit to a portion of data types.
- For example we can define a general template called `Test<any_data_type>`, a partial specialization called `Test<any_data_type*>` that applies to pointers and a specialization `Test<const char*>` that applies to `const char*` exclusively.
- Program example:

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

//general, justice for all type:-)
template <class any_data_type>
any_data_type Test(any_data_type Var1)
{return Var1;}

//partial specialization for all pointers type
template <class any_data_type>
any_data_type * Test(any_data_type *Var2)
{return Var2;}

//specialization, just for const char *
template <>
const char * Test(const char *Var3)
{return Var3;}

//do some testing
int main()
{
	int p = 5;
	//calls Test(any_data_type
	int q = Test(p);
	double r = Test(3.1234);

	cout<<"General types = "<<q<<endl;
	cout<<"General types = "<<r<<endl;

	//calls Test(any_data_type*)
	int *s = Test(&p);
	char *t = "Partial lor!";
	cout<<"Partial types = "<<s<<endl;
	cout<<"Partial types = "<<t<<endl;

	//calls Test(const char *)
	const char *u = Test("Specialized!");
	cout<<"Specialization type = "<<u<<endl;
	return 0;
}
```

Output:



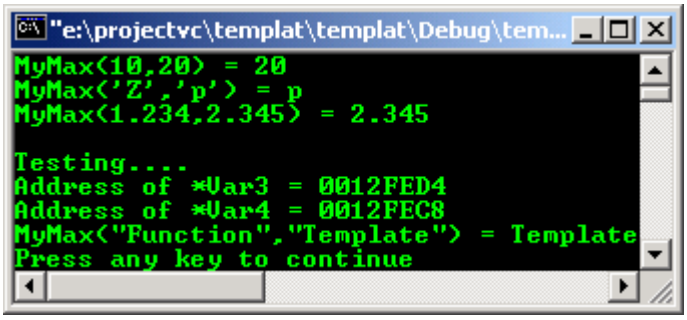
24.7.3 Template Function Specialization

- As for class template, function template also has specialization.
- Program example:

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

template <class any_data_type>
any_data_type MyMax(any_data_type Var1, any_data_type Var2)
{
	return Var1 > Var2 ? Var1:Var2;
```

```
}

//specialization of MyMax() for char *
template<>
char* MyMax(char* Var3, char* Var4)
{
	return strcmp(Var3,Var4)> 0 ? Var3:Var4;
}

int main()
{
	cout<<"MyMax(10,20) = "<<MyMax(10,20)<<endl;
	cout<<"MyMax('Z','p') = "<<MyMax('Z','p')<<endl;
	cout<<"MyMax(1.234,2.345) = "<<MyMax(1.234,2.345)<<endl;
	char* Var3 = "Function";
	char* Var4 = "Template";
	cout<<"\nTesting...."<<endl;
	cout<<"Address of *Var3 = "<<&Var3<<endl;
	cout<<"Address of *Var4 = "<<&Var4<<endl;

	cout<<"MyMax(\"Function\",\"Template\") = "<<MyMax(Var3,Var4)<<endl;
	return 0;
}
```

Output:



24.8 typename Keyword

- You may encounter this keyword somewhere, sometime. The keyword `typename` is used to specify the identifier that follows is a type.
- In other word, `typename` keyword tells the compiler that an unknown identifier is a type. Consider the following example:

```
template <class any_data_type>
class MyClass
{
	typename any_data_type::another_data_type * ptr;
	//...
};

int main()
{
	return 0;
}
```

- Here, `typename` is used to clarify that `another_data_type` is a type of class `any_data_type`. Thus, `ptr` is a pointer to the type `any_data_type::another_data_type`. Without `typename`, `another_data_type` would be considered a static member. Hence,

```
any_data_type::another_data_type * ptr
```

- Would be interpreted as a multiplication of **value** `another_data_type` of type `any_data_type` with `ptr`.
- According to the qualification of `another_data_type` being a type, any type that is used in place of `any_data_type` must provide an inner type of `another_data_type`. For example, the use of type `Test` as a template argument

```
MyClass<Test> x;
```

- Is possible only if type `Test` has an inner type definition such as the following:

```
class Test
{
      typedef int another_data_type;
      ...
};
```

- In this case, the `ptr` member of `MyClass<Test>` would be a pointer to type `int`. However, the `another_data_type` could also be an abstract data type such as a class as shown below.

```
class Test
{
      class another_data_type;
      ...
};
```

- `typename` is always necessary to qualify an identifier of a template as being a **type**, even if an interpretation is not a type.
- Thus, the general rule in C++ is that any **identifier of a template** is considered to be a **value**, except it is qualified by `typename` keyword, then it is a **type**.
- Apart from this, `typename` can also be used instead of class in a template declaration:

```
template <typename any_data_type>
class MyClass
{};
```

- Well, now you may be ready to create your own template or just proceed to the next Modules, see how these templates used to construct the C++ headers and how to use other readily available class or function templates in your programs.
- The following is a program example compiled using **g++** on Fedora 3 machine.

```
//*******template.cpp**********
#include <iostream>
#include <string>
//for strcmp()
#include <cstring>
using namespace std;

//primary template, for all type
template <class any_data_type>
any_data_type MyMax(const any_data_type Var1, const any_data_type Var2)
{
       cout<<"Primary template..."<<endl;
       return Var1 < Var2 ? Var2 : Var1;
}

//specialization for const char *, empty parameter list
template <>
const char *MyMax(const char *Var1, const char *Var2)
{
       cout<<"Specialization..."<<endl;
       //comparison for const char *
       return (strcmp(Var1, Var2)<0) ? Var2 : Var1;
}

//do some testing
int main()
{
       //call primary
       int Highest = MyMax(7, 20);
       //call primary
       char p = MyMax('x', 'r');
       string Str1 = "Class", Str2 = "Template";
       //call primary
       string MaxStr = MyMax(Str1, Str2);

       cout<<"The bigger between 7 and 20 is "<<Highest<<endl;
```

```
        cout<<"The bigger between 'x' and 'r' is "<<p<<endl;
        cout<<"The bigger between \""<<Str1<<"\" and \""<<Str2<<"\" is
        "<<MaxStr<<"\n\n";

        //call specialization
        const char *Var3 = "Class";
        const char *Var4 = "Template";
        const char *q = MyMax(Var3, Var4);
        cout<<"The bigger between \""<<Var3<<"\" and \""<<Var4<<"\" is "<<q<<endl;
        return 0;
}
```

[bodo@bakawali ~]$ g++ template.cpp -o template
[bodo@bakawali ~]$ ./template

```
Primary template...
Primary template...
Primary template...
The bigger between 7 and 20 is 20
The bigger between 'x' and 'r' is x
The bigger between "Class" and "Template" is Template

Specialization...
The bigger between "Class" and "Template" is Template
```

----------o0o----------

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.