Templates in C++

Helping Manual

Adopted from CS304 Handouts by VU

Generic Programming

Generic programming refers to programs containing generic abstractions (general code that is same in logic for all data types like printArray function), then we instantiate that generic program abstraction (function, class) for a particular data type, such abstractions can work with many different types of data.

Advantages

Major benefits of this approach are:

a. Reusability: Code can work for all data types.

b. Writability: Code takes lesser time to write.

c. Maintainability: Code is easy to maintain as changes are needed to be made in a single function or class instead of many functions or classes.

Templates

In C++ generic programming is done using templates. Templates are of two kinds,

a. **Function Templates** (in case we want to write general function like printArray)
b. **Class Templates** (in case we want to write general class like Array class)

Function Templates

A function template can be parameterized to operate on different types of data types.

Declaration:
We write template keyword above any function make any function as template function, they can be declared in any one of the following ways,

```
template< class T >
void funName(T x);

// OR
template< typename T >
void funName(T x);

// OR
template< class T, class U, ... >
void funName(T x, U y, ...);

template< typename T >
void printArray(T* array, int size)
{
	for (int i = 0; i < size; i++)
		cout << array[i] <<","; // here data type of array is T
}
```

Template function will be instantiated for a particular data type according to passed argument as shown below**,**

```
int main() {
int iArray[5] = { 1, 2, 3, 4, 5 };
void printArray(iArray, 5); // Instantiated for int[] as passed array is of type int []
char cArray[3] = { ‘a’, ‘b’, ‘c’ };
void printArray(cArray, 3); // Instantiated for char[] as argument is of type
char[]
return 0;
}
```

Explicit Type Parameterization:

In case a function template does not have any parameter then we have to explicitly mention the data type for which we want to create that function as shown below,

```
template <typename T>
T getInput() {
	T x;
	cin >> x;
```

```
	return x;
}

int main() {
	int x;
	x = getInput(); // Error!
	double y;
	y = getInput(); // Error!
	return 0;
}

int main() {
	int x;
	x = getInput< int >(); //will now work
	double y;
	y = getInput< double >(); //will now work
	return 0;
}
```

Template Specializations

A template compiler generated code may not handle all the types successfully; in that case we can give explicit specializations for a particular data type(s). For example suppose we have written a function isEqual(…, …) that compares two values of data type and return true or false depending upon the values are equal or not,

```
template< typename T >
bool isEqual(T x, T y) {
	return (x == y);
}
```

isEqual (6,6) should return true
isEqual (6,7) should return false
isEqual (6.6,6.6) should return true
isEqual (6.5,6.6) should return false
isEqual ('A','A') should return true
isEqual ('A','a') should return false

Until here the function will work correctly but consider the statement below,
isEqual ("abc","xyz")

This is instantiation of isEqual function for built in type char [] or char *14, this function will fail to give correct result simply because we have given its implementation as

return (x == y); So here it will be translated by compiler in,
return (char * == char *); or return (char [] == char []);

So for char* datatypes, we will specialize the function as:

```
template< >
bool isEqual< const char* >(
	const char* x, const char* y) {
	return (strcmp(x, y) == 0);
}
```

Partial Specialization

The above example is an example of "**complete specialization"**. Following partial specialization of this function deals with pointers to objects,

```
template< typename T >
bool isEqual( T* x, T* y ) {
return ( *x == *y );
}
```

Multiple Type Arguments

```
template< typename T, typename U >
T my_cast(U u) {
	return (T)u; // U type will be converted to T type and will be returned
}

int main() {
	double d = 10.5674;
	int j = my_cast(d); //Error
	int i = my_cast<int>(d); // need to explicity mention about type of T (int in this
case) as it is used only For
	//return type not as parameter

	return 0;
}
```

Class Template

Class template provides functionality to operate on different types of data and in this way facilitates reuse of classes.

We can definition a class template as follows:

- template< class T > class Xyz { … };
- template< typename T > class Xyz { … };

Member Templates

Member functions of a template class implicitly become functions templates; they work for instantiations (int, char, float, double so on…) of that class, however there are some situations where we need explicit template functions in our class taking more template parameters other than one implicit template parameter (parameter given to this class as parameter while creating its object). A class or class template can have member functions that are themselves templates

```
template<typename T>
class Complex {
	T real, imag;
public:
	// Complex<T>( T r, T im )
	Complex(T r, T im) :
		real(r), imag(im) {}
	// Complex<T>(const Complex<T>& c)
	Complex(const Complex<T>& c) :
		real(c.real), imag(c.imag) {}
	…
};
```

Now see that main function for this class in which we are assigning Complex class float instance to double, it will result in an error,

```
int main() {
	Complex< float > fc(0, 0);
	Complex< double > dc = fc; // Error
	return 0;
}
```

In order for the above main function to work correctly, we need to change the implementation of our Complex class as follows:

```
template<typename T> class Complex {
	T real, imag;
public:
	Complex(T r, T im) :
		real(r), imag(im) {}
	template <typename U>
	// this copy constructor is now taking two template parameters one implicit T
	// and other explicit U
	Complex(const Complex<U>& c) :
		real(c.real), imag(c.imag) {}
	…
};
```

Class Template Specialization

Consider a class template:

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

template <class T>
class List
{
private:
	int size;
```

```
	T* ptr;
public:
	List(int size = 5);
	void insert(T const& data, int index);
	List(const List<T>&);

};
template <typename T>
List<T>::List(int size)
{
	this->size = size;
	this->ptr = new T[size];
}



template <typename T>
List<T>::List(const List<T>& that)
{
	this->ptr = new T[that.size];
	this->size = that.size;
	for (int i = 0; i < this->size; i++)
	{
		this->ptr[i] = that.ptr[i];
	}
}

template <typename T>
void List<T>::insert(T const& data, int index)
{
	if (index<0 || index>this->size)
		exit(EXIT_FAILURE);

	else this->ptr[index] = data;

}
```

Now consider the main function:

```
int main() {
	List< char* > list1(2);
	char names[][10] = { "ali", "ahmed" };
	list1.insert(names[0],0); //shallow copy
	list1.insert(names[1],1); //shallow copy
	List< char* > list2(list1); //shallow copy

}
```

To solve the above problem, we need to write a specialized template class:

```
template <>
class List<char*>
{
private:
	int size;
	char** ptr;
public:
```

```
        List(int size = 5);
        void insert(char* const& data, int index);
        List(const List<char*>&);

};


List<char*>::List(int size)
{
        this->size = size;
        this->ptr = new char*[size];
        for (int i = 0; i < this->size; i++)
        {
                this->ptr[i] = nullptr;
        }

}

List<char*>::List(const List<char*>& that)
{
        this->size = that.size;
        this->ptr = new char*[size];
        this->ptr = new char*[size];
        for (int i = 0; i < this->size; i++)
        {
                this->ptr[i] = new char[strlen(that.ptr[i])];
                strcpy(this->ptr[i], that.ptr[i]);
        }
}

void List<char*>::insert(char* const& data, int index)
{
        if (index < 0 || index >= this->size)
                exit(EXIT_FAILURE);

        if (this->ptr[index] == data)
                return;

        if (this->ptr[index] != nullptr)
        {
                delete this->ptr[index];
        }

        this->ptr[index] = new char[strlen(data)];
        strcpy(this->ptr[index], data);

}
```

Note: instead of specializing class template, we could also specialize member templates (It will also work fine):

```
template<>
List<char*>::List(int size)
{
        this->size = size;
        this->ptr = new char*[size];
        for (int i = 0; i < this->size; i++)
```

```
        {
                this->ptr[i] = nullptr;
        }

}
```