Chapter 16: Templates

CSE 2010
Week 14

Background

When writing a program, there may be a need to apply the same code to different data types.

Suppose we need to write a program that finds the smallest between 2 values. These values can vary in data type.

We can easily accomplish this with function overloading.

Example: Find the smaller of two values

```
#include <iostream>
using namespace std;
//functions to find the smaller between 2 values
char smaller(char first, char second){
        if(first<second)
                return first;
        else
                return second;
}
int smaller(int first, int second){
        if(first<second)
                return first;
        else
                return second;
}
double smaller(double first, double second){
        if(first<second)
                return first;
        else
                return second;
}
int main(){
        cout <<"Smaller of 'a' and 'B': "<<smaller('a','B')<<"\n";
        cout <<"Smaller of 12 and 15: "<<smaller(12,15)<<"\n";
        cout <<"Smaller of 44.2 and 33.1: "<<smaller(44.2,33.1)<<"\n";
        return 0;
}
```

Output:

```
Smaller of 'a' and 'B': B
Smaller of 12 and 15: 12
Smaller of 44.2 and 33.1: 33.1
```

- This works fine. Depending on the data types that are provided in the function call, the compiler will select the appropriate function.
- But what if we also wanted this to work for other data types like strings, floats, etc?
- We would have to implement additional functions for each data type.
- Is there a better way?

Templates!

- We need a way to *generalize* our functions and programs, which will allow us to easily reuse them in several special cases.
- We want to abstract away the differences, and keep the parts that are the same.

Literal definition of a template:
“A preset format for a document or file, used so that the format does not have to be recreated each time it is used.”

Templates in C++:
A tool that allows a single function or class to work with a variety of data types.

- A template allows a function or a class definition to be *parameterized* by type, instead of values.
- For this chapter we will learn about template functions & template classes.

Template Functions

Syntax of a template function :

```
template<typename type_var1, …, typename type_varn>
return_type function_name(parameters)
{
    //statements
}
```

1. To define a template function, we first put the keyword ***template***.
2. This is followed by a list of the type parameters, which is surrounded by angle brackets.
   - **`typename`** `type_var1, …,` **`typename`** `type_varn` is used to list the number of generic types your function will need.
3. The return type can be a generic type or regular data type.

Template Functions - Example

Syntax of a template function :

```
template<typename type_var1, …,typename type_varn>

return_type function_name(parameters)

{
        //statements
}
```

Let’s run this and see what happens

```
/*
 * Filename: smaller.cpp
 * Example program that uses a template function
 * to find smaller of 2 values
 */
#include <iostream>
using namespace std;

template <typename T>
T smaller(T first, T second){
        //return the smaller of the two
        if(first < second)
                return first;
        else
                return second;
}

int main(){
        cout << "Smaller of 'a' and 'B': " << smaller('a','b') << "\n";
        cout << "Smaller of 12 and 15: " << smaller (12,15) << "\n";
        cout << "Smaller of 44.2 and 33.1: " << smaller(44.2, 33.1) << "\n";
        cout << "Smaller of 'cse' and 'csE' is: " << smaller ("cse","csE") << "\n";

        return 0;
}
```

Template Function Instantiation

Templates are efficient not just because of their ease in implementation, but also in their execution.

The polymorphism of templates (defining the necessary nontemplate functions) occurs during **compilation time**, not run time.

This means that when a program invoking a function template is compiled, the compiler creates only as many versions of the function as needed by the function calls.

This process is referred to as ***template instantiation***.

Swapping Two Values

```cpp
/*
 * Filename: swapping.cpp
 * Program that uses a template function to swap values
 */

#include <iostream>
using namespace std;

template<typename T>
void swapping(T& first, T& second){
        T temp = first;
        first = second;
        second = temp;
}

int main(){
        //swapping int values
        int num1 = 6, num2 = 70;
        cout << "Before swapping: " << num1 << " " << num2 << "\n";
        swapping(num1,num2);
        cout << "After swapping: " << num1 << " " << num2 << "\n\n";

        //swapping strings
        string s1 = "cat", s2 = "dog";
        cout << "Before swapping: " << s1 << " " << s2 << "\n";
        swapping(s1,s2);
        cout << "After swapping: " << s1 << " " << s2 << "\n";

        return 0;

}
```

Output:

```
Before swapping: 6 70
After swapping: 70 6

Before swapping: cat dog
After swapping: dog cat
```

Common Error: Invalid Type Parameters

The arguments you call a function with need to be appropriate for the template function in terms of amount and type.

```
template <typename T>
T smaller(T first, T smaller);
```

- Yes you can send any datatype to the function, BUT the datatype needs to be the same for the arguments.

Error:

```
cout << smaller (23, 67.2) << "\n";// error! Two different types for T
```

We can avoid this error if we explicitly convert the arguments during the call.

```
cout << smaller <double> (23, 67.2) << "\n";// 23 will be sent as 23.0
```

Template Function Overloading

Just like regular function overloading, we can can overload a function template to have several functions with the same name but different parameters.

Let’s look at an example! (smallest.cpp)

Printing array and vector elements

We can overload a print template function that will print either array or vector elements.

Recall that when we use arrays, their size is determined at compilation time, so every array has a int size associated with it (we will call this N).

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

//function to accept vectors
template<typename T>
void print(T v){
        cout << "Using vector template function\n";
        cout << "Vector elements:";
        for(int i = 0; i < v.size(); i++){
                cout << v[i] << " ";
        }
        cout << "\n";
}
template<typename T, int N>
void print(T(&arr)[N]){
        cout << "Using array template function\n";
        cout << "Array elements:";
        for(int i = 0; i < N;i++)
        {
                cout << arr[i]<< " ";
        }
        cout << "\n";
}
int main()
{
        int a[5] = {2,4,6,8,10};
        double a2[3] = {76.7,56.4,2.5};
        int a3[] = {1,2,3};
        vector<int> v1{1,3,5,7,9};
        vector<double> v2{25.4,34.23,88.1};

        print(a);
        print(a2);
        print(a3);
        cout << "\n";
        print(v1);
        print(v2);
        return 0;
}
```

Output:

```
Using array template function
Array elements:2 4 6 8 10
Using array template function
Array elements:76.7 56.4 2.5
Using array template function
Array elements:1 2 3

Using vector template function
Vector elements:1 3 5 7 9
Using vector template function
Vector elements:25.4 34.23 88.1
```

Class Templates

We have learned in previous chapters that a class is a combination of data members and member functions.

Now consider that we need a class with the same data members and overall functionality, but with different data types.

We can accomplish this with a class template!

Syntax:

```
template <typename T>//you can have multiple generic types
class className
{
      private:
            T data;

      public:
            className(T init); //overload constructor
            T get() const;//accessor
            void set(T d);//mutator

};
```

Syntax for function definitions:

```
template <typename T>
className<T>::className(T init):data(init)
{
}

template <typename T>
T className <T>::get()const
{
      return data;
}

template <typename T>
void className <T>::set(T d)
      data = d;
}
```

Compilation of Class Templates

- Templates are not like ordinary classes in the sense that the compiler doesn’t generate object code for a template or any of its members until the template is instantiated with concrete types.
- Acceptable methods of compiling class templates vary depending on the C++ compiler you use, but the following method should work across all versions.
- **The inclusion method**
  - Define your template class in a .h file
  - Define your template class functions in a .cpp file
  - Include the .cpp file in whatever program file you are using the class in.
  - When you go to compile, you only need to compile the program file, not the class template.

Let’s look at an example: Pair Class