Comp151

Function Templates & Class Templates
Revised Version

Function & Class Templates

In this lesson we will learn more about function and class templates. We will learn

- Funtion template definition
- Funtion template instantiation
- Funtion template argument matching
- Funtion templates with more than one argument
- Class templates
- Difference between function and class templates

Function Templates


```
int max(const int& a, const int& b) { ... }
string max(const string& a, const string& b) { ... }
```

- We often find a set of functions that look very much alike, e.g. for a certain type `T`, the `max` function has the form

  ```
  T max(const T& a, const T& b){ ... }
  ```

- In C++, we can define one single function definition using templates:

  ```
  template <typename T>
  T max(const T& a, const T& b)
  {
      return (a > b) ? a : b;
  }
  ```

Function Template ..

The **typename** keyword may be replaced by **class**; the following template definition is equivalent to the one above:

```
template <class T>
T max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

The above template definitions are not functions; You cannot call a function template.

Function Template Instantiation

The compiler *creates* functions using function templates.

```
int i = 2, j = 3;
cout << max(i, j);

string a("Hello"), b("World");
cout << max(a, b);
```

- In this case, the compiler creates two *different* `max()` functions using the function template

  ```
  template<typename T> T max(const T& a, const T& b);
  ```

- This is called template instantiation. The parameter `T` in the template definition is called the formal parameter or formal argument.
- In the above code, the template is instantiated with the actual arguments `int` and `string`, respectively.

Template: Formal Argument Matching

- When the compiler instantiates a template, it determines the actual type of the template parameter by looking at the types of the actual arguments:

```
template <typename T> T max(const T& a, const T& b);

void f() {
    cout << max(3, 5);          // T is int;
    cout << max(4.3, 5.6);      // T is double
}
```

- However, there is **NO** automatic type conversion for template arguments:

```
cout << max(4, 5.5); // Error
```

- You can help by *explicitly instantiating* the function template:

```
cout << max<double>(4, 5.5);
```

Template: More Than One Formal Argument

Another way of using the `max` template with arguments of different types is changing its definition in the following way:

```
template <typename T1, typename T2>
T1 max(const T1& a, const T2& b)
{
    return (a > b) ? a : b;
}

void f() {
    cout << max(4, 5.5);        // T1 is int, T2 is double
    cout << max(5.5, 4);        // T1 is double, T2 is int
}
```

However, there is a subtle problem here; the return type of `max` is the same as the type of the first argument. So what will the above code print?

The following template definition does not suffer from this problem:

```
template <typename T1, typename T2>
void print_max(const T1& a, const T2& b)
{
    cout << ((a > b) ? a : b) << endl;
}

void f()
{
    print_max(4, 5.5);                // Prints 5.5
    print_max(5.5, 4);                // Prints 5.5
}
```

Template Arguments: Too Many Combinations

Consider the following code:

```
short s = 1; char c = 'A';
int i = 1023; double d = 3.1415;

print_max(s, s); print_max(s, c);
print_max(c, s); print_max(s, i);
// ... And all other combinations; 16 in total.
```

- The compiler should instantiate `print_max()` for 16 different combinations of arguments.
- With the current compiler technology, this means that we get 16 (almost identical) fragments of code in the executable program. There is no sharing of code.
- So, an innocent looking program may have a surprisingly large binary size, if you are not careful.

Class Templates

The template mechanism works for classes as well. In the next slide we will see a *thing* class template.

The lines

```
thing<int> i_thing(5);
thing<string> s_thing("COMP151");
```

will create two objects of *totally different* classes.

i_thing is an object of class `thing<int>`
s_thing is an object of class `thing<string>`.

The objects are each called with a constructor that takes one parameter

Class Templates

```
template <typename T>
class thing
{ public:
   thing(T data);
   ~thing() {delete x;};
   T get_data() {return *x;};
   T set_data(T data);
private:
   T * x; };

template <typename T>
thing<T>:: thing(T data)
 {x = new T; *x = data;}

template <typename T>
T thing<T>::set_data(T data)
 {*x = data;}
```

```
#include<iostream.h>
#include<string>

main()
{ thing<int> i_thing(5);
 thing<string> s_thing("COMP151");

 cout << i_thing.get_data() << endl;
 cout << s_thing.get_data() << endl;

 i_thing.set_data(10);
 s_thing.set_data("CPEG");
 cout << i_thing.get_data() << endl;
 cout << s_thing.get_data() << endl;
}
```

Class Templates

The class template format is particularly suitable for container classes. As an example, the C++ *vector* is actually a class template: Note that, in the example $\mathbf{I}$ and $\mathbf{S}$ are not of the same class so we can not write $\mathbf{I} = \mathbf{S}$.

Class Templates


```
#include<vector>                                           // stl vector class
#include<stdio.h>
#include<string>
void main()
{ vector<int> I, J;
  vector<string> S;

  I.push_back(10); I.push_back(20);
  J.push_back(5); J.push_back(15); J.push_back(25);
  S.push_back("COMP"); S.push_back("CPEG");

  cout << "I: " << I[0] << " " << I[1] << endl;
  cout << "J: " << J[0] << " " << J[1] << " " << J[2] << endl;
  cout << "S: " << S[0] << " " << S[1] << endl;

  I = J;
  cout << "I: " << I[0] << " " << I[1] << " " << I[2] << endl;
  cout << "J: " << J[0] << " " << J[1] << " " << J[2] << endl;

  I[1] = 17;
  cout << "I: " << I[0] << " " << I[1] << " " << I[2] << endl;
  cout << "J: " << J[0] << " " << J[1] << " " << J[2] << endl;

  I = S;                                                   // Illegal!
}
```

Difference Between Class & Function Templates

Remember that for function templates, the compiler can deduce the template arguments:

```
int i = max(4, 5);
int j = max<int>(7, 2); // OK, but not needed
```

For class templates, you always have to specify the actual template arguments; the compiler does *not* deduce the template arguments:

```
vector primes;      // Error
primes.push_back(2);
```

Function Template: Common Errors


```
template <class T> T* create() { ... } ;

template <class T> void f()
{
    T a;
    ...
}
```

e.g. With the template definition above, what is the function type of the following calls?

```
create();      // Error! Use e.g. create<int>(); instead
f();           // Error! Use e.g. f<float>(); instead
```

Reason: the compiler has to be able to deduce the actual function types from calls to the template function.

Separate Compilation For Templates??

For normal (non-template) functions, we usually put the declaration in a header file, and the definition in the corresponding *.cpp file. According to the C++ standard, we would expect this to work for function templates as well:

```
// File "max.h"
template <typename T> T max(const T& a, const T& b);

// File "max.cpp"
template <typename T> T max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

The same should apply to separating class definition in a *.h and implementation of class member functions in the corresponding *.cpp.

But a function/class is instantiated *only if* it is used *and* possibly on *every* use . . . ???

Can We Do This?

```
// File "max.h"
template <typename T> T max(const T& a, const T& b);

// File "max.cpp"
template <typename T> T max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

```
// File "a.cc"
#include "max.h"
int main() { cout << max(2,4) << endl; }
```

```
g++ -c a.cc; g++ -c max.cc; g++ *.o
```

Inclusion Compilation of Templates


```
// File "a.cc"
#include "max.h"

int main()
{
    cout << max(2,4) << endl;
}
```

```
// File "max.h"

template <typename T>
T max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

```
g++ -o a.out a.cc
```

While there are other ways to compile templates, the simplest one is to include the template header file in <u>every</u> files which use the template.