COMP2012H

Function Templates & 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;
}
```

- Prefer the `typename` syntax (the `class` syntax is more old-fashioned).
- The above <u>template definitions</u> 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 <u>template instantiation</u>. The parameter T in the template definition is called the <u>formal parameter</u> or <u>formal argument</u>.
- In the above code, the template is <u>instantiated</u> with the <u>actual arguments</u> `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 {() {
    cout << max(3, 5);              // T is int
    cout << max(4.3, 5.6);          // T is double
}
```

  * This ability to automatically infer the type of variables is known as <u>type inference</u>. (This approach was pioneered by functional languages with parametric polymorphism, like ML and Haskell, which are called <u>type-inferring languages</u>.)

- However, there is **<u>no</u>** 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?

Template: More Than One Formal Argument ..

- 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 “Code Bloat”: 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(s, i); print_max(s, d);
print_max(c, s); print_max(c, c); print_max(c, i); print_max(c, d);
print_max(i, s); print_max(i, c); print_max(i, i); print_max(i, d);
print_max(d, s); print_max(d, c); print_max(d, i); print_max(d, d);
```

- 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

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

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 template mechanism works for classes as well. This is particularly useful for defining container classes.

  In the next few slides we will define

  ```
  template<typename T>
  class List_Node
  ```

  and a *container class*

  ```
  template<typename T>
  class List
  ```

  that uses `List_Node`.
- Note the line `friend class list<T>` in the definition of `List_Node`.

Class Templates: List_Node

```
template<typename T>
class List_Node {
public:
    List_Node(const T& data);
    List_Node<T>* next();
    // Other member functions

private:
    List_Node<T>* next_m;
    List_Node<T>* prev_m;
    T data_m;

    friend class List<T>;
};
```

Class Templates: List_Node

```
template<typename T>
List_Node<T>::List_Node(const T& data)
    : data_m(data), next_m(0), prev_m(0) { }

template<typename T>
List_Node<T>* List_Node<T>::next()
{
    return next_m;
}
```

Class Templates: List

- Using the `List_Node<T>` class, we define our list class template:

```
template<typename T>
class List {
public:
    List();
    void append(const T& item);
    // Other member functions
private:
    List_Node<T>* head_m;
    List_Node<T>* tail_m;
};
```

Class Templates: List

```
template<typename T>
List<T>::List() : head_m(0), tail_m(0) { }

template<typename T>
void List<T>::append(const T& item)
{
   List_Node<T>* new_node = new List_Node<T>(item);
   if (! tail_m) {
      head_m = tail_m = new_node;
   } else {
      // ...
   }
}
```

Class Templates: List Example

- Now we can use our brand new list class to store any type of element that we want, without having to resort to "code reuse by copying":

```
List<Person> people;
Person person("Jane Doe");
people.append(person);
people.append(Person("John Doe"));

List<int> primes;
primes.append(2);
primes.append(3);
primes.append(5);
```

Difference between class & function templates

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

```
List primes;                      // Error
primes.append(2);
```

Function Template: Common Errors

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

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

- With the template definition above, what is the function type of the following calls?

```
create();
f();
```

Function Template: Common Errors

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

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

- 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 infer 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.hpp"
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;
}
```

- It seems like the same should apply to separating class definition in a *.hpp 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.hpp"
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.cpp"
#include "max.hpp"
int main() { cout << max(2,4) << endl; }
```

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

Inclusion Compilation of Templates

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

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

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

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