Lecture 6 : Template

Acknowledgement : courtesy of Prof. Dekai Wu lecture slides

Function Template

- 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

```
int max(const int& a, const int& b) { …}
string max(const string& a, const string& b) { …}
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 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.
- There is no automatic type conversion for template arguments

```
cout<< max(3, 5);// T is int
cout<< max(4.3, 5.6); // T is double
cout<< max(4, 5.5); // Error
cout<< max<double>(4,5.5); // OK
```

More than one argument

```
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
}
```

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>;
};

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;
};

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("JaneDoe");
people.append(person);
people.append(Person("JohnDoe"));

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
List<int> primes; // OK
primes.append(2);
```

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.

Exercise

- Write a stack class with templates

```
template<typename T>
class Stack {
public:
    Stack(int = 100);
    ~Stack();
    bool push(const T&);
    bool pop(T&);
    bool isEmpty() const;
    bool isFull() const;

private:
    int size;
    int top;
    T *stackPtr;
};
```

```
int main()
{
    int x, y;

    Stack<int> intStack(5);
    intStack.push(5);
    intStack.push(8);
    intStack.pop(x);
    intStack.pop(y);

    cout << x << endl;
    cout << y << endl;

    return 0;
}
```