C++ Templates

Template and Generics in C++

- Generic programming lets us write classes and functions that are polymorphic across unrelated types at compile time.
- Examples of generic types in C++ include:
  - Sequential Containers
    - vector
    - listLike OOP, generic programming relies on the form of polymorphism.
    - queue
    - Stack
    - …
  - Associative Container
    - map
    - Set
  - Generic Algorithms
    - find
    - replace

Template Functions

Template Functions Definition

- Template function is a generic function that can be instantiated for different signatures.
- Template functions will be defined similar to other functions, proceeding with keyword template following with <parameter list>.
- Parameter list may have one or more types.

Template Functions Definition

- Each formal parameter must be preceded by the "**class**" or "**typename**" keyword:
- Example:

  template <**class** T, **class** V, **typname** K> fun (T, V, K);
- The name of a formal parameter can be reused across template function definition or prototype.
- Each parameter **must appear at least** once in the signature.
- The name of formal parameter need not to be the same across forward declaration and function definition.

Template Functions Definition

- A template function can be declared extern, inline, or static:
  - The following function prototype is correct;

```
template <class T>
extern T min (T , T);
```

  - The following function prototype is wrong;

```
extern template <class T>
 T min (T , T);
```

Template Function Example

```
template <class T>
void swap (T* a, T*  b);  // function prototype

template <class T>
void swap (T* a, T* b)
{
    T temp ;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Template Functions Instantiation

```
struct person { int age; char name[30];};
int main() {
        int n = 8, m = 6;
        float x = 4.5, y =5.5;
        Person s = {45, "Jack Moore"};
        Person t = {40, "Russ Lewis"};
        swap (&n,&m); // instantiation to swap(int*, int*)
        swap (&x, &y); // instantiation to swap(float*, float*)
        swap (&s,&t);   // instantiation to swap(person*,
                        // person*)
        return 0;
}
```

Template Functions Overloading

- A template function can be overloaded provided that the signature of each instance can be distinguished either by argument type or number:

```
template < class type>
Type sum(Type, int);

template < class type>
Type sum(Type, Type);

template < class type>
Type sum(Type);
```

Template Function Specialization

- There are cases that either the general template expansion is inappropriate or inefficient for a particular type:

```
template <class T>
T min (T t1, T t2)
{
    return (t1<t2) ? t1 : t2;
}

// specialization of function min

char* min (char* s1, char* s2)
{
    return strcmp(s1,s2) < 0 ? s1: s2;
}
```

- **A complete example and further detail will be discussed during the lecture**

Template Classes

Template Classes

- Templates are used to create a generic class and can be used to create containers such as queues, stacks, linked lists, vectors and etc.
- Generally, the template and non template classes behave essentially the same.

Template Class Definition

```
template <class T>
class Vector
{
   public:
      Vector(int s);
      ~Vector();
      T getValue(int elem);
      void display();
   private:
      T *array;
      int size;
};
```

Template Class Member Function

```
template <class T>
Vector<T>::Vector (int s) {
    array = new T[s];
    assert (array !=0);
    size = s;
    }

template <class T>
T Vector<T>::getValue (int elem) {
    return array[elem];
    }
```

Templates Class Definition

- Once the template class has been made known to the program, it can be used as a type specifier the same as a non template class. However, the only difference is that, use of a template class name must always include its parameter list enclosed by angle brackets (except within its own class definition).

Template Class Instantiation

- A template class is instantiated by appending the full list of actual parameters enclosed by angle brackets to the template class name.

  Vector <float> v (50);

  - Creates a vector of 50 floats

  Vector <double> * vptr = new Vector<double>(100)

  - Creates a pointer pointing to a vector object, containing 100 elements of.

- Same as regular classes, template class can have a forward declaration. Here are couple of examples:

  template <class T> class QueueItem;

  template <class T1, class T2, class T3> class Container;

Template Class Specialization

- Sometime, there are particular types for which a default member function is not sufficient. In this cases you can provide an explicit implementation to handle a particular type.
- Similar to template functions one can provide the specialization of of a template class.
- Further detail and an example will be discussed during the lectures.

Friend Template Operator Functions in a Template Class

- A friend operator function must be declared in a template class using the following format:
- ***friend return_type operator operator_symbol <type_list> (parm_list)***

- *Example:*

```
template class <K, D>
class LookupTable{
  friend ostream& operator << <K, D> (ostream& os, LookupTable<K, D>  &  lt);
  …
  …
};
```

Template Compilation Modes

- In Standard C++ for the ordinarily cases, the compiler needs only to see the declaration of the functions and definition of the classes. Therefore, the declaration of functions and the definition of the classes should go into the “header file” and the the implementation of functions and member functions into the “.cpp file”.
- For the template case this is different. The compiler needs to see the the implementation as well.
- Therefore, three compilation models are possible:
  - Single Header Model
  - Inclusion Compilation Model
  - Separate Compilation Model
    - not recommended. only some compilers support this one

Single Header Model

```
// File name: test.h
#ifndef _TEST
#define _TEST
template <class T> Queue {
 public:
     void fun(T x);
   …
};  // end of class definition


template<class T>
Queue<T>::void fun(T x)
{
   …
   …
}

#endif
```

Inclusion Compilation Model

```
// File name: test.h
#ifndef _TEST
#define _TEST
template <class T> void fun(T);
…
…
#include test.cpp
#endif

// File name: test.cpp
template<class T> void fun(T x) {
    …
    …
}
```