Templates

Function templates

- Function templates are special functions that can operate with *generic types*.
- This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.
- In C++ this can be achieved using *template parameters*.
-

Templates parameters

- A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function.
- These function templates can use these parameters as if they were any other regular type.

Template declaration

The format for declaring function templates with type parameters is:

template <class identifier> function_declaration;

template <typename identifier> function_declaration;

The only difference between both prototypes is the use of either the keyword class or the keyword typename.

Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

Template example

To create a template function that returns the greater one of two objects we could use:

```
template  <class myType>
myType  GetMax (myType a, myType b) {
return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter.

This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type.

As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

Using template function

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

int x,y;
GetMax <int> (x,y);

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Using template function- Example


```
template <class T> T GetMax (T a, T b) {
    T result;
   result = (a>b)? a : b;
   return (result);
}
int main () {
   int i=5, j=6, k;
   long l=10, m=5, n;
   k=GetMax<int>(i,j);
   n=GetMax<long>(l,m);
   cout << k << endl;
   cout << n << endl;
   return 0;
}
```

```
6
10
```

Example (cont.)


```
template <class T> T max(T a, T b) {
   return a > b ? a : b ;
 }
void main() {
   cout << "max(10, 15) = " << max(10, 15) << endl ;
   cout << "max('k', 's') = " << max('k', 's') << endl ;
   cout << "max(10.1, 15.2) = " << max(10.1, 15.2) <<
   endl ; }
```

Program Output

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
```

Example 2


```
template<class T> class Key {
   T k;
   T* kptr;
   int length;
   public:
         Key(T);
         // ...
};
```

Suppose the following declarations appear later:

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

Example 3


```
template<class T>
        void quicksort(T a[], int leftarg, int rightarg) {
   if (leftarg < rightarg) {
        T pivotvalue = a[leftarg];
        int left = leftarg - 1;
        int right = rightarg + 1;
        for(;;) {
            while (a[--right] > pivotvalue);
            while (a[++left] < pivotvalue);
            if (left >= right) break;
            T temp = a[right];
            a[right] = a[left];
            a[left] = temp;
        }
         int pivot = right;
        quicksort(a, leftarg, pivot);
        quicksort(a, pivot + 1, rightarg);
   }
```

- the type of the array to be sorted, T
- the name of the array to be sorted, a
- the lower bound of the array, leftarg
- the upper bound of the array, rightarg

Example 3 (cont.)


```
int main(void) {
   int sortme[10];
   for (int i = 0; i < 10; i++) {
         sortme[i] = rand();
         cout << sortme[i] << " ";
    }
   cout << endl;
   quicksort<int>(sortme, 0, 10 - 1);
   for (int i = 0; i < 10; i++)
   cout << sortme[i] << " ";
   cout << endl;
   return 0;
}
```

```
4086 16212 7419 23010 5627 31051 17515 10113 5758 16838
31051 23010 17515 16838 16212 10113 7419 5758 5627 4086
```

Class template

Exist the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T> class mypair {
        T values [2];
        public:
                mypair (T first, T second) {
                        values[0]=first;
                        values[1]=second;
                }
  };
```

Using class template

The class above defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

mypair<int> myobject (115, 36);

This same class would also be used to create an object to store any other type:

mypair<double> myfloats (3.0, 2.18);

Class template example


```
template <class T> class mypair {
   T a, b;
   public:
   mypair (T first, T second) {a=first; b=second;}
   T getmax ();
};
template <class T>
T mypair<T>::getmax () {
   T retval  = a>b? a : b;
   return retval;
}
int main () {
   mypair <int> myobject (100, 75);
   cout << myobject.getmax();
return 0; }
```

100

- How many T's? There are three T's in this declaration:
- The first one is the template parameter.
- The second T refers to the type returned by the function.
- And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Member functions of class templates

You may define a template member function outside of its class template definition.

```
template<class T> class X {
public:
   T operator+(T);
 };
template<class T> T X<T>::operator+(T arg1) {
   return arg1;
};
int main() {
   X<char> a;
   X<int> b;
   a +'z';
   b + 4;
}
```

- The overloaded addition operator has been defined outside of class X.
- The statement a + 'z' is equivalent to a.operator+('z').
- The statement b + 4 is equivalent to b.operator+(4).

Example 2


```
template <class T> class Stack {
public:
   Stack(int = 10) ;
   ~Stack() { delete [] stackPtr ; }
   int push(const T&);
   int pop(T&) ;
   int isEmpty()const { return top == -1 ; }
   int isFull() const { return top == size - 1 ; }
private:
   int size ; // number of elements on Stack.
   int top ;
   T* stackPtr ;
} ;
```

Example 2(cont.)


```
//constructor with the default size 10
template <class T> Stack<T>::Stack(int s) {
   size = s > 0 && s < 1000 ? s : 10 ; top = -1 ; // initialize stack
   stackPtr = new T[size] ;
 }
// push an element onto the Stack
template <class T> int Stack<T>::push(const T& item) { if
   (!isFull()) {
         stackPtr[++top] = item ;
         return 1 ; // push successful
   }
   return 0 ; // push unsuccessful
 }
```

Example 2(cont.)


```
// pop an element off the Stack
template <class T> int Stack<T>::pop(T& popValue) { if
   (!isEmpty()) {
         popValue = stackPtr[top--] ;
         return 1 ; // pop successful
   }
   return 0 ; // pop unsuccessful
}
```

Example 2 - Implementation


```
void main() {
   typedef Stack<float> FloatStack ;
   FloatStack fs(5) ;
   float f = 1.1 ;
   cout << "Pushing elements onto fs" << endl ;
   while (fs.push(f)) {
         cout << f << ' ' ; f += 1.1 ;
   }
   cout << endl << "Stack Full." << endl << endl << "Popping
   elements from fs" << endl ;
   while (fs.pop(f))
   cout << f << ' ' ; cout << endl << "Stack Empty" << endl ;
   cout << endl ;
```

```
Pushing elements onto fs
1.1 2.2 3.3 4.4 5.5
Stack Full.
Popping elements from fs
5.5 4.4 3.3 2.2 1.1
Stack Empty
```

Friends and templates

- There are four kinds of relationships between classes and their friends when templates are involved:
  - One-to-many: A non-template function may be a friend to all template class instantiations.
  - Many-to-one: All instantiations of a template function may be friends to a regular non-template class.
  - One-to-one: A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
  - Many-to-many: All instantiations of a template function may be a friend to all instantiations of the template class.

Class Template	friend declaration in class template X	Results of giving friendship
template class <T> class X	friend void f1() ;	makes f1() a friend of all instantiations of template X. For example, f1() is a friend of X<int>, X<A>, and X<Y>.
template class <T> class X	friend void f2(X<T>&) ;	For a particular type T for example, float, makes f2(X<float>&) a friend of class X<float> only. f2(x<float>&) cannot be a friend of class X<A>.
template class <T> class X	friend A::f4() ; // A is a user defined class with a member function f4() ;	makes A::f4() a friend of all instantiations of template X. For example, A::f4() is a friend of X<int>, X<A>, and X<Y>.

template class <T> class X	friend C<T>::f5(X<T>&) ; // C is a class template with a member function f5	For a particular type T for example, float, makes C<float>::f5(X<float>&) a friend of class X<float> only. C<float>::f5(x<float>&) cannot be a friend of class X<A>.
template class <T> class X	friend class Y ;	makes every member function of class Y a friend of every template class produced from the class template X.
template class <T> class X	friend class Z<T> ;	when a template class is instantiated with a particular type T, such as a float, all members of class Z<float> become friends of template class X<float>.

Example of friends and templates


```
class B{
   template<class V> friend int j();
}

template<class S> g();

template<class T> class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
   template<class U> friend int h();
};
```

Function g() has a one-to-one relationship with class A

Function h() has a many-to-many relationship with class A. Function h() is a function template. For all instantiations of A all instantiations of h() are friends.

For example, g<int> is a friend of A<int>.

Function j() has a many-to-one relationship with class B.

Template specialization

- If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.
- For example, let's suppose that we have a very simple class called *my container* that can store one element of any type and that it has just one member function called *increase*, which increases its value.
- But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase.

Class template specialization


```
//class template:
template <class T> class mycontainer {
  T element;
  public:
        mycontainer (T arg) {element=arg;}
        T increase () {return ++element;}
};
```

Class template specialization


```
template <> class mycontainer <char> {
   char element;
   public:
         mycontainer (char arg) {element=arg;}
         char uppercase () {
            if ((element>='a')&&(element<='z'))    element+='A'-'a';
            return element;
         }
};
int main () {
   mycontainer<int> myint (7);
   mycontainer<char> mychar ('j');
   cout << myint.increase() << endl;
   cout << mychar.uppercase() << endl;
   return 0;
}
```

```
8
J
```

Example 2


```
template <class T> T m
  return a > b ? a : b ;
}
int main() {
  cout << "max(10, 15)
  cout << "max('k', 's')
  cout << "max(\"Alad
  max("Aladdin", "Jasm
  return 0 ;
}
```

Not quite the expected results! Why did that happen?
The function call max("Aladdin", "Jasmine") causes the compiler to generate code for max(char*, char*), which compares the addresses of the strings!
To correct special cases like these or to provide more efficient implementations for certain types, one can use template specializations.

max(10, 15) = 15
max('k', 's') = s
max("Aladdin", "Jasmine") = Aladdin

Template Class Partial Specialization


```
//base template class
template<typename T1, typename T2>
class X { } ;
//partial specialization
template<typename T1>
class X<T1, int> { } ;
int main() {
// generates an instantiation from the base template
   X<char, char> xcc ;
//generates an instantiation from the partial specialization
   X<char, int> xii ;
   return 0 ;
}
```

Implicit instantiation

- Unless a template specialization has been explicitly instantiated or explicitly specialized, the compiler will generate a specialization for the template only when it needs the definition. This is called implicit instantiation.
- If the compiler must instantiate a class template specialization and the template is declared, you must also define the template.
- For example, if you declare a pointer to a class, the definition of that class is not needed and the class will not be implicitly instantiated

Example of implicit instantiation


```
template<class T> class X {
public:
   X* p;
   void f();
   void g();
};
X<int>* q;
X<int> r;
X<float>* s;
r.f(); s->g();
```

- The compiler requires the instantiation of the following classes and functions:
- X<int> when the object r is declared
- X<int>::f() at the member function call r.f()
- X<float> and X<float>::g() at the class member access function call s->g()
- Therefore, the functions X<T>::f() and X<T>::g() must be defined in order for the above example to compile

Explicit instantiation

- You can explicitly tell the compiler when it should generate a definition from a template. This is called explicit instantiation.
- Explicit instantiation declaration syntax

Examples of explicit instantiation


```
template<class T> class Array {
   void mf();
};
template class Array<char>; // explicit instantiation
template void Array<int>::mf(); // explicit instantiation
```

```
template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&); // explicit instantiation
```

Examples 2


```
int* p = 0;
template<class T> T g(T = &p);
template char g(char); // explicit instantiation
```

```
template <class T> class X {
private:
   T v(T arg) { return arg; };
};
template int X<int>::v(int); // explicit instantiation
```