



1 Template functions

Consider the following `Swap()` function which swaps two integers:

```
1 void Swap(int& a, int& b) {int t=a; a=b; b=t;}
```

Assume that we need to define similar functions for other data types:

```
1 void Swap(double& a, double& b) {double t=a; a=b; b=t;}
2 void Swap(char& a, char& b) {char t=a; a=b; b=t;}
```

The drawback of this approach is rewriting the same code with very minor modifications, namely changing the data types.

C++ provides a mechanism for code generation that enables the programmer to avoid rewriting the same code for different data types:

```
1 // Swap.h
2 template<class Type>
3 void Swap(Type& a, Type& b) {Type t=a; a=b; b=t;}
```

By including the above **template function** in a **.cpp** file that includes an **implicit instantiation** of the template function as follows:

```
1 #include "Swap.h"
2 int main()
3 {
4     int a=2, b=5;
5     Swap(a, b); // Implicit instantiation of the Swap<int> template function
6     cout<<a<<" "<<b<<endl; // Prints: 5 2
7     return 0;
8 }
```

The statement `Swap(a, b);` is an implicit instantiation of the `Swap()` template function that takes `int`. The compiler will then look at the `Swap.h` file and generates the following code:

```
1 void Swap(int& a, int& b) {int t=a; a=b; b=t;}
```

It is safe to include the `Swap.h` file in several `.cpp` files, and make several implicit instantiations. Multiple implicit instantiations will never cause the compiler to instantiate the same code twice for the same data type.

In some situations, the compiler might not be able to detect the data type. In such cases, a statement like `Swap<int>(a, b);` should be used instead of `Swap(a, b);`.

```

1 #include "Swap.h"
2 int main()
3 {
4     int a=2, b=5;
5     Swap(a, b); // Implicit instantiation of Swap<int> template function
6     cout<<a<<" "<<b<<endl; // Prints: 5 2
7     Swap<int>(a, b); // Already instantiated from the above line
8     cout<<a<<" "<<b<<endl; // Prints: 2 5
9     double x=2.3, y=5.1;
10    Swap<double>(x, y); // Implicit instantiation of Swap<double> function
11    cout<<x<<" "<<y<<endl; // Prints: 5.1 2.3
12    return 0;
13 }
```

The above code will cause the compiler to generate the following functions:

```

1 void Swap(int& a, int& b) {int t=a; a=b; b=t;}
2 void Swap(double& a, double& b) {double t=a; a=b; b=t;}
```

In order to reduce the compilation time, it is possible to separate the template definition into `.h` and `.cpp` files. In that case, implicit instantiations will not work by including `Swap.h` file. Only **explicit instantiations** in `Swap.cpp` file will work. An explicit instantiation for a specific data type must occur at most once.

```

1 // Swap.h
2 template<class Type>
3 void Swap(Type&, Type&);
```

```

1 // Swap.cpp
2 template<class Type>
3 void Swap(Type& a, Type& b) {Type t=a; a=b; b=t;}
4 // Explicit instantiations
5 template void Swap<int>(int&, int&);
6 template void Swap<char>(char&, char&);
```

```

1 #include "Swap.h"
2 int main()
3 {
4     int a=2, b=5;    double x=2.3, y=5.1;
5     Swap<int>(a, b); ✓
6     Swap<double>(x, y); ✗ // Can not do implicit instantiation
7                           // since Swap.h does not contain the definition
8 }
```

Since the definition of the `Swap()` function uses the copy constructor and the assignment operator of the data type `Type`, `Swap()` can work with any data type that supports these operations. It can not be instantiated for other data types.

A template function may contain several templated data types. Also, template functions can be overloaded:

```

1  template<class Type>
2  void Out(Type a, char c) {cout<<c<<" "<<a<<endl;}
3  // Explicit instantiation
4  template void Out<int>(int, char);
5
6  template<class TA, class TB>
7  void Out(TA a, TB b, char c) {cout<<c<<" "<<a<<" "<<b<<endl;}
8  // Explicit instantiation
9  template void Out<double, int>(double, int, char);
10
11 int main()
12 {
13     Out<int>(4, 'p'); // Prints: p 4
14     Out<double, int>(3.5, 9, 'w'); // Prints: w 3.5 9
15     Out<int, int>(3, 9, 'q'); // Implicit instantiation, Prints: q 3 9
16     Out(3, 9.5, 'z'); // Implicit instantiation of Out<int, double>, Prints: z 3 9.5
17     Out(4.2, 'r'); // Implicit instantiation of Out<double>, Prints: r 4.2
18 }
```

It is possible to replace the word `class` by the word `typename`, which is actually more readable, since the data type is not required to be a class. It can be any built-in or struct data type.

```

1  template<typename Type>
2  void Out(Type a, char c) {cout<<c<<" "<<a<<endl;}
```

Instead of taking unspecified data types, a template function can take a specific data type and the compiler generates code for each particular value:

```

1  template<int A>
2  void Out(char c) {cout<<A<<c;}
3  // Explicit instantiation
4  template void Out<3>(char); // generates: void Out(char c) {cout<<3<<c;}
5
6  int main()
7  {
8     Out<3>('p'); // Prints: 3p
9     Out<4>('q'); // Implicit instantiation of Out<4>, Prints: 4q
10 }
```

The most common used data types for this purpose are `int` and pointers to functions. It is not possible to use `double` or `float` data types.

2 Template classes

Template classes behave similarly to template functions. If the template class takes unspecified data types, the compiler will generate a separate class for each combination of instantiated data types. If the template class takes specific data types, the compiler will generate a separate class for each combination of instantiated values. A template class or template function may contain both unspecified data types and specific data types as follows:

```

1 // MyClass.h
2 template<class TA, class TB, int C>
3 class MyClass
4 {
5     TA a;   TB b;
6 public:
7     void Set(TA _a, TB _b) {a=_a; b=_b;}
8     void Show() {cout<<a<<b<<C<<endl;}
9 };

```

```

1 // MyClass.cpp
2 #include "MyClass.h"
3 // Explicit instantiation
4 template class MyClass<int, double, 3>;

```

The explicit instantiation above generates the following code:

```

1 class MyClass
2 {
3     int a;   double b;
4 public:
5     void Set(int _a, double _b) {a=_a; b=_b;}
6     void Show() {cout<<a<<b<<3<<endl;}
7 };

```

The following file makes use of `MyClass` template class:

```

1 #include "MyClass.h"
2 extern template class MyClass<int, double, 3>;
3 int main()
4 {
5     MyClass<int, double, 3> m; // Use explicit instantiation
6     m.Set(1, 2.5);
7     m.Show();                 // Prints: 1 2.5 3
8     MyClass<char, int, 7> u; // Implicit instantiation
9     u.Set('a', 3.3);
10    u.Show();                 // Prints: a 3.3 7
11 }

```

3 Template specialization

It is possible to provide different implementations of a template if we have some information about the unknown template parameters. The compiler will select the most specialized implementation (the one with fewest unknowns) matching the given parameters. Class templates can be partially or totally specialized, while function templates can be only totally specialized (a specialization of a template function must eliminate all unknowns). The following example illustrates template specialization:

```
1  template<class TA, class TB>
2  class MyClass
3  {public: void Show() {cout<<"Different"<<endl;}};
4
5  template<class T> // partial specialization
6  class MyClass<T,T>
7  {public: void Show() {cout<<"Similar"<<endl;}};
8
9  template<class TA, class TB> // partial specialization
10 class MyClass<TA*,TB>
11 {public: void Show() {cout<<"FirstPointer"<<endl;}};
12
13 template<class TA> // partial specialization
14 class MyClass<TA,double>
15 {public: void Show() {cout<<"SecondDouble"<<endl;}};
16
17 template<> // total specialization
18 class MyClass<int,double>
19 {public: void Show() {cout<<"IntDouble"<<endl;}};
20
21 template<int a>
22 void Go() {cout<<a<<endl;}
23
24 template<> // total specialization
25 void Go<7>() {cout<<"seven"<<endl;}
26
27 int main()
28 {
29     MyClass<int, char> m1; m1.Show(); // Prints: Different
30     MyClass<int, int> m2; m2.Show(); // Prints: Similar
31     MyClass<int*, char> m3; m3.Show(); // Prints: FirstPointer
32     MyClass<char, double> m4; m4.Show(); // Prints: SecondDouble
33     MyClass<int, double> m5; m5.Show(); // Prints: IntDouble
34     Go<2>(); // Prints: 2
35     Go<7>(); // Prints: seven
36 }
```

4 Designing with templates

Templates are a powerful design tool. In a previous lecture we needed to write separate `Game` classes for each `Hero` type. To avoid code repetition, we wrote one `Game` class and included a `Hero*` as a member variable in class `Game` that can be assigned to address of any object of class derived from `Hero`. With templates, we can ask the compiler to generate separate `Game` classes, each class aggregates an object from a concrete class derived from `Hero`.

```

1 template<class ConcreteHero>
2 class Game
3 {
4     ConcreteHero hero;
5 public:
6     void Play() {hero.Jump();}
7 };
8 class SuperMario {public: void Jump() {/*SuperMario jumps*/}}
9 class SuperMan {public: void Jump() {/*SuperMan jumps*/}}
10 class BatMan {public: void Jump() {/*BatMan jumps*/}}
11
12 int main()
13 {
14     Game<SuperMan> g1; g1.Play();
15     Game<BanMan> g2; g2.Play();
16 }
```

Another example, class `Button` has two derived classes `MetalButton` and `WoodButton`. Class `Animation` has two derived classes `SkyAnimation` and `SeaAnimation`. Suppose we need to create `AnimatedButton` class which combines properties of `Button` and `Animation`. Inheriting it from `Button` and aggregating `Animation` enables using classes derived from `Animation`, but does not enable using classes derived from `Button`. Aggregating both classes has the drawback that `AnimatedButton` is no more derived from `Button` and can not be used in its place.

Templates overcome these drawbacks as follows:

```

1 template<class CButton, class CAnimation>
2 class AnimatedButton: public CButton, public CAnimation {};
```

This solution enables us to use derived classes from both `Button` and `Animation`, such as `AnimatedButton<WoodButton, SkyAnimation>`. Actually, they need not derive from these classes, but they need to define the functions used in the template. Moreover, objects of class `AnimatedButton` can be used in place of `Button` and `Animation` since they are inherited from both classes. Polymorphism can not provide such flexibility.

Templates are more efficient than polymorphism, because templates do not have the run-time overheads of checking the object type and directing execution to the intended function. But, templates are not useful in some situations such as defining an array of base class pointers that are able to hold the addresses of different derived classes. In this case, polymorphism is essential.