Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Programming-2 CS213

Templates

Dr. Amin Allam

1 Template functions

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

```
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:

```
void Swap(double& a, double& b) {double t=a; a=b; b=t;}
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:

```
// Swap.h
template<class Type>
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 instatiation of the template function as follows:

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

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

```
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);`.

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

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

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

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

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

```
#include "Swap.h"
int main()
{
    int a=2, b=5;   double x=2.3, y=5.1;
    Swap<int>(a, b); ✔
    Swap<double>(x, y); ✖ // Can not do implicit instantiation
                          // since Swap.h does not contain the definition
}
```

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:

```
template<class Type>
void Out(Type a, char c) {cout<<c<<" "<<a<<endl;}
// Explicit instantiation
template void Out<int>(int, char);

template<class TA, class TB>
void Out(TA a, TB b, char c) {cout<<c<<" "<<a<<" "<<b<<endl;}
// Explicit instantiation
template void Out<double, int>(double, int, char);

int main()
{
    Out<int>(4, 'p'); // Prints: p 4
    Out<double, int>(3.5, 9, 'w'); // Prints: w 3.5 9
    Out<int, int>(3, 9, 'q'); // Implicit instantiation, Prints: q 3 9
    Out(3, 9.5, 'z'); // Implicit instantiation of Out<int, double>, Prints: z 3 9.5
    Out(4.2, 'r'); // Implicit instantiation of Out<double>, Prints: r 4.2
}
```

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.

```
template<typename Type>
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:

```
template<int A>
void Out(char c) {cout<<A<<c;}
// Explicit instantiation
template void Out<3>(char);     // generates: void Out(char c) {cout<<3<<c;}

int main()
{
    Out<3>('p'); // Prints: 3p
    Out<4>('q'); // Implicit instantiation of Out<4>, Prints: 4q
}
```

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 sperate 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:

```
// MyClass.h
template<class TA, class TB, int C>
class MyClass
{
    TA a;  TB b;
public:
    void Set(TA _a, TA _b) {a=_a; b=_b;}
    void Show() {cout<<a<<b<<C<<endl;}
};
```

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

The explicit instantiation above generates the following code:

```
class MyClass
{
    int a;  double b;
public:
    void Set(int _a, double _b) {a=_a; b=_b;}
    void Show() {cout<<a<<b<<3<<endl;}
};
```

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

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

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:

```cpp
template<class TA, class TB>
class MyClass
{public: void Show() {cout<<"Different"<<endl;}};

template<class T>   // partial specialization
class MyClass<T,T>
{public: void Show() {cout<<"Similar"<<endl;}};

template<class TA, class TB>   // partial specialization
class MyClass<TA*,TB>
{public: void Show() {cout<<"FirstPointer"<<endl;}};

template<class TA>   // partial specialization
class MyClass<TA,double>
{public: void Show() {cout<<"SecondDouble"<<endl;}};

template<>   // total specialization
class MyClass<int,double>
{public: void Show() {cout<<"IntDouble"<<endl;}};

template<int a>
void Go(){cout<<a<<endl;}

template<> // total specialization
void Go<7>(){cout<<"seven"<<endl;}

int main()
{
    MyClass<int, char> m1; m1.Show();   // Prints: Different
    MyClass<int, int> m2; m2.Show();   // Prints: Similar
    MyClass<int*, char> m3; m3.Show();   // Prints: FirstPointer
    MyClass<char, double> m4; m4.Show();   // Prints: SecondDouble
    MyClass<int, double> m5; m5.Show();   // Prints: IntDouble
    Go<2>();   // Prints: 2
    Go<7>();   // Prints: seven
}
```

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`.

```cpp
template<class ConcreteHero>
class Game
{
    ConcreteHero hero;
public:
    void Play() {hero.Jump();}
};
class SuperMario {public: void Jump(){/*SuperMario jumps*/}}
class SuperMan {public: void Jump(){/*SuperMan jumps*/}}
class BatMan {public: void Jump(){/*BatMan jumps*/}}

int main()
{
    Game<SuperMan> g1; g1.Play();
    Game<BanMan> g2; g2.Play();
}
```

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:

```cpp
template<class CButton, class CAnimation>
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.