Templates

Computer Programming for Engineers
(DASF003-41)

Instructor:
Sungjae Hwang (jason.sungjae.hwang@gmail.com)

Recap

- **Inheritance**
  - Base/Derived class
  - Redefining member functions
  - Ctors/Dtors
- **Polymorphism**
  - virtual (Overriding)
  - override, final
  - Late(Dynamic) binding
  - Slicing problem
- **Operator Overloading**
  - Global overloading
  - Overloading as member functions

Today

Function Templates

- Syntax, defining
- Compiler complications

Class Templates

- Syntax
- Example: array template class

TEMPLATE BASICS

Introduction

C++ templates

- Allow very "general" definitions for functions and classes
- Type names are "parameters" instead of actual types
- Precise definition determined **at run-time**

Instances of C++ templates

- Instance of function template ➔ function
- Instance of class template ➔ class
  - Instance of class ➔ object

Function Templates vs. Overloading

Swap Function:

```
void swapValues( int& var1, int& var2 )
{
   int temp = var1;
   var1 = var2;
   var2 = temp;
}
```

- Applies only to variables of type int

What about other types?

Function Templates vs. Overloading

- Could overload function for chars:

```
void swapValues(char& var1, char& var2)
{
   char temp = var1;
   var1 = var2;
   var2 = temp;
}
```

- Recall: Advantage of function overloading?

Function Templates vs. Overloading

- Could overload function for chars:

```
void swapValues(char& var1, char& var2)
{
   char temp = var1;
   var1 = var2;
   var2 = temp;
}
```

- Recall: Advantage of function overloading?

```
int main(){
 swapValues(1,2);
 swapValues('a','b');
}
```

C++

VS

```
int main(){
 swapValues_i(1,2);
 swapValues_c('a','b');
}
```

C

Function Templates vs. Overloading

Can we improve the code further?

Function Templates vs. Overloading

- Could overload function for chars:

```
void swapValues(int& var1, int& var2){
   int temp = var1;
   var1 = var2;
   var2 = temp;
}
void swapValues(char& var1, char& var2){
   char temp = var1;
   var1 = var2;
   var2 = temp;
}
```

- But notice: code is nearly identical!
  - Only difference is type used in 3 places

Function Template Syntax

- Allow "swap values" of any type variables:

```
template<class T>
void swapValues( T& var1, T& var2 )
{
   T temp = var1;
   var1 = var2;
   var2 = temp;
}
```

- First line called "template prefix"
  - Tells compiler what's coming is "template"
  - "**class**" here indicates "**type**", and T is a type parameter
  - See the next pages for more details.

Template Prefix

- **Recall:**

```
template<class T>
```

  - In this usage, "class" means "type", or "classification"

- **Can be confused with other "known" use of word "class"!**
  - C++ allows keyword "`typename`" in place of keyword "`class`" here

```
template<typename T>
```

  - Use whichever you want

Template Prefix

- **Again:**

```
template<class T>
```

- **T can be replaced by any type**
  - Predefined or user-defined (like a C++ class type)
- **In function definition body:**
  - T used like any other type
- **Note: can use other than "T", but T is "traditional" usage**

Calling a Function Template

- Consider following call:

```
template<class T>
void swapValues( T& var1, T& var2 ){
   T temp = var1;
   var1 = var2;
   var2 = temp;
}
```

```
swapValues( int1, int2 );
```

  - C++ compiler "generates" function definition for two `int` parameters using template
- Likewise for all other types
- Need not do anything "special" in call
  - Required definition **automatically generated by compiler.**

Template function, swap


```
 1 #include <iostream>
 2 using namespace std;
 3
 4 template<class T>
 5 void swapValues(T& variable1, T& variable2){
 6   T temp;
 7
 8   temp = variable1;
 9   variable1 = variable2;
10   variable2 = temp;
11 }
12
13 int main(){
14   int integer1=1, integer2=2;
15   cout << "Original integers " << integer1 << " " << integer2 << endl;
16
17   swapValues(integer1, integer2);
18   cout << "Swapped integers " << integer1 << " " << integer2 << endl;
19
20   char char1='A', char2='B';
21   cout << "Original chars " << char1 << " " << char2 << endl;
22
23   swapValues(char1, char2);
24   cout << "Swapped chars " << char1 << " " << char2 << endl;
25
26   return 0;
27 }
```

Another Function Template

- Declaration/prototype:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3);
```

- Definition:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
   cout << stuff1 << endl
      << stuff2 << endl
      << stuff3 << endl;
}
```

showStuff Call

- Consider function call:

```
showStuff(2, 3.3, 4.4);
```

- Compiler generates function definition
  - Replaces T with double
    - Since second parameter is type double
- Displays:

```
2
3.3
4.4
```

Compiler Complications

- **Function declarations and definitions**
  - Typically we have them separate
  - For templates → not supported on most compilers!

- **Safest to place template function definition in file where invoked**
  - Many compilers require it appear first
  - Often we #include all template definitions

More Compiler Complications

- **Check your compiler's specific requirements**
  - Some need to set special options
  - Some require special order of arrangement of template definitions vs. other file items

- **Most usable template program layout:**
  - Template definition in same file it's used
  - Ensure template definition precedes all uses
    - Can `#include` it

MORE ABOUT TEMPLATES

Multiple Type Parameters

Can have:

```
template<class T1, class T2>
```

Not typical

- Usually only need one "replaceable" type
- Cannot have "unused" template parameters
  - Each must be "used" in definition
  - Error otherwise!
    - If not used, the compiler cannot deduce the types.
    - Code for the function cannot be generated.
    - The function call routine cannot find a proper definition.

Template function, showStuff


```
 1 #include <iostream>
 2 using namespace std;
 3
 4 template<class T>
 5 void showStuff1(int stuff1, T stuff2, T stuff3){
 6   cout << stuff1 << endl;
 7   cout << stuff2 << endl;
 8   cout << stuff3 << endl;
 9 }
10
11 template<class T1, class T2>
12 void showStuff2(int stuff1, T1 stuff2, T2 stuff3){
13   cout << stuff1 << endl;
14   cout << stuff2 << endl;
15   cout << stuff3 << endl;
16 }
17
18 #if 0
19 template<class Z1, class Z2>
20 void showStuff3(int stuff1, Z1 stuff2, Z1 stuff3){
21   cout << stuff1 << endl;
22   cout << stuff2 << endl;
23   cout << stuff3 << endl;
24 }
25 #endif
26
27 int main(){
28   int integer1=1, integer2=2, integer3=3;
29   showStuff1(integer1,integer2,integer3);
30
31   cout << "==========" << endl;
32
33   double d1 = 1.1;
34   char c1 = 'A';
35
36   showStuff2(integer1, d1, c1);
37
38   //showStuff3(integer1, d1, d1);
39
40   return 0;
41 }
```

Inappropriate Types in Templates

- **Can use any type in template for which code makes "sense"**
  - Code must behave in appropriate way
  - e.g., `swapValues()` template function
    - Cannot use type for which assignment operator isn't defined
    - Example: an array:

```
int a[10], b[10];
swapValues(a, b);
```

    - Arrays cannot be "assigned"!

Template function, inappropriate type


```
 1 #include <iostream>
 2 using namespace std;
 4 template<class T>
 5 void swapValues(T& variable1, T& variable2){
 6   T temp;
 7
 8   temp = variable1;
 9   variable1 = variable2;
10   variable2 = temp;
11 }
13 int main(){
14   int integer1=1, integer2=2;
15   cout << "Original integers " << integer1 << " " << integer2 << endl;
16
17   swapValues(integer1, integer2);
18   cout << "Swapped integers " << integer1 << " " << integer2 << endl;
19
20   char char1='A', char2='B';
21   cout << "Original chars " << char1 << " " << char2 << endl;
22
23   swapValues(char1, char2);
24   cout << "Swapped chars " << char1 << " " << char2 << endl;
25
26   int a[5] = {1,2,3,4,5};
27   int b[5] = {6,7,8,9,10};
28
29   cout << "Original arrays " << a << " " << b << endl;
30
31   swapValues(a, b);
32   cout << "Swapped arrays " << a << " " << b << endl;
33
34   return 0;
35 }
```

`swapValues` for Arrays

Passing arrays to functions

- When an array is passed as follows, its type is set to a **pointer**!

```
void foo(int a[3])
{
  for(auto &i : a) // ERROR! No begin() with pointer type
    cout << &i << endl;
}
```

- We need to pass an array with the following syntax.

```
void foo(int (&a)[3])
{
  for(auto &i : a)
    cout << &i << endl;
}
```

Generalizing `swapValues`

- `void foo(int (&a)[3])`
  - We can use this function only for an array of three integer element.
  - For generalizing
    - Replace int with template class T
    - Replace 3 with the size N

```
template<class T, int N>
void foo(T (&a)[N])
{
  for(auto &i : a)
    cout << &i << endl;
}
```

We can use non-type parameters for templates. In this example, **size_t N** is a **non-type parameter**.

    - Constant integer
    - Enumeration type
    - Function pointer & pointer

Non-type parameter


```
template<class T>
void swapValues( T& var1, T& var2 )
{
  cout << "swapValues template" << endl;
  T temp = var1;
  var1 = var2;
  var2 = temp;
}

template<class T, size_t N>
void swapValues( T (&var1)[N], T (&var2)[N] )
{
  cout << "swapValues template for arrays" << endl;
  for(int i=0; i<N; i++) {
    T temp = var1[i];
    var1[i] = var2[i];
    var2[i] = temp;
  }
}
```

Defining Templates Strategies

- **Develop function normally**
  - Using actual data types
- **Completely debug "ordinary" function**
- **Then convert to template**
  - Replace type names with type parameter as needed
- **Advantages:**
  - Easier to solve "concrete" case
  - Deal with algorithm, not template syntax

CLASS TEMPLATES

Class Templates

Can also "generalize" classes

```
template<class T>
```

- Can also apply to class definition
- All instances of "T" in class definition replaced by type parameter
- Just like for function templates!

Once template defined, can declare objects of the class

Class Template Definition

Display 16.4 Class Template Definition (1 of 2)

```
template<class T>
class Pair
{
public:
   Pair();
   Pair(T firstVal, T secondVal);
   void setFirst(T newVal);
   void setSecond(T newVal);
   T getFirst() const;
   T getSecond() const;
private:
   T first; T second;
};
```

class here has different meaning from the below line. You can use **typename** for better understandings.

Class Template Definition

- Display 16.4 Class Template Definition (2 of 2)

```
template<class T>
Pair<T>::Pair(T firstVal, T secondVal)
{
   first = firstVal;
   second = secondVal;
}

template<class T>
void Pair<T>::setFirst(T newVal)
{
   first = newVal;
}
```

Template Class Pair

- Objects of class have "pair" of values of type T
- Can then declare objects:

```
Pair<int> score;
Pair<char> seats;
```

  - Objects then used like any other objects

- Example uses:

```
score.setFirst(3);
score.setSecond(0);
```

Pair Member Function Definitions

- **Notice in member function definitions:**
  - Each definition is itself a "template"
  - Requires template prefix before each definition
  - Class name before :: is "`Pair<T>`"
    - Not just "Pair"

Class Templates as Parameters

Consider:

```
int addUP(const Pair<int>& thePair);
```

- The type (int) is supplied to be used for T in defining this class type parameter
- It "happens" to be call-by-reference here

Again:

- template types can be used anywhere standard types can

Class Templates Within Function Templates

- Rather than defining new overload:

```
template<class T>
T addUp(const Pair<T>& thePair);
// precondition: Operator + is defined for values of type T
// returns sum of two values in thePair
```

- Function now applies to all kinds of numbers

Restrictions on Type Parameter

- **Only "reasonable" types can be substituted for T**
- **Consider:**
  - Assignment operator must be "well-behaved"
  - Copy constructor must also work
  - If T involves pointers, then destructor must be suitable!

- **Similar issues as function templates**

Class Template Usages


```
1 #include <iostream>
 2 using namespace std;
 3
 4 class IntCount{
 5    int count;
 6 public:
 7    IntCount() { count = 0; }
 8    int getCount() { return count; }
 9    void setCount(int val) { count = val; }
10 };
11
12 class DoubleCount{
13    double count;
14 public:
15    DoubleCount() { count = 0; }
16    double getCount() { return count; }
17    void setCount(double val) { count = val; }
18 };
19
```

```
20 //template <typename T>
21 template <class T>
22 class Counter{
23    T count;
24 public:
25    Counter() { count = 0; }
26    T getCount() { return count; }
27    void setCount(T val) { count = val; }
28 };
29
30 int main(){
31   Counter <int> intCnt;
32   Counter <double> doubleCnt;
33   return 0;
34 }
```

Templates and Inheritance

- **Nothing new here**
- **Derived template classes**
  - Can derive from template or non-template class
  - Derived class is then naturally a template class
- **Syntax same as ordinary class derived from ordinary class**

Template class inheritance (1)


```
 1 #include <iostream>
 2 using namespace std;
 3
 4 template <class T>
 5 class Parent{
 6   T val;
 7   public:
 8     Parent(T arg1) { val = arg1; }
 9     void print() { cout << val << endl; }
10 };
11
12 class Child : public Parent<int> {
13   public:
14     Child(int a) : Parent<int>(a){}
15 };
16
17 int main(){
18   Child obj1(10);
19   obj1.print();
20   return 0;
21 }
```

Template class inheritance (2)


```
 1 #include <iostream>
 2 using namespace std;
 4 template <class T>
 5 class Parent{
 6   T val;
 7   public:
 8     Parent(T arg1) { val = arg1; }
 9     void print() { cout << val << endl; }
10 };
12 template <class Z>
13 class Child : public Parent<Z> {
14   public:
15     Child(Z a) : Parent<Z>(a){}
16 };
18 int main(){
19   Child<int> obj1(10);
20   Child<double> obj2(10.20);
21
22   obj1.print();
23   obj2.print();
24   return 0;
25 }
```

Type Definitions: typedef or using (C++11)

- Can define new "class type name"
  - To represent specialized class template name
- Example:

```
typedef int myint
typedef Pair<int> PairOfInt;
using PairOfFloat = Pair<float>;

// objects of type Pair<int>:
PairOfInt pair1, pair2;
PairOfFloat fpair1, fpair2;
```

```
template <class T>
using PairOfNums = Pair<T>;
typedef Pair<int> PairOfInt;

PairOfInt pair1, pair2;
PairOfNums<float> fpair1;
```

- Name can also be used as parameter, or anywhere else type name allowed

Using keyword


```
 1 #include <iostream>
 2 using namespace std;
 3
 4
 5 class IntCount{
 6    int count;
 7 public:
 8    IntCount() { count = 0; }
 9    int getCount() { return count; }
10    void setCount(int val) { count = val; }
11 };
12
13 class DoubleCount{
14    double count;
15 public:
16    DoubleCount() { count = 0; }
17    double getCount() { return count; }
18    void setCount(double val) { count =
val; }
19 };
20
```

```
21 //template <typename T>
22 template <class T>
23 class Counter{
24    T count;
25 public:
26    Counter() { count = 0; }
27    T getCount() { return count; }
28    void setCount(T val) { count = val; }
29 };
30
31 template <class V>
32 using PairOfNums = Counter<V>;
33
34 int main(){
35   Counter <int> intCnt;
36   Counter <double> doubleCnt;
37
38   PairOfNums<int> intCnt2;
39   intCnt2.setCount(30);
40   cout << intCnt2.getCount() << endl;
41   return 0;
42 }
```

Summary

- Function template
- Class template
- Inheriting class template
- Using vs typedef