Templates

Dr. Sanem Sarıel Talay

OOP – Templates

- The template feature in C++ provides a way to **reuse *source* code**.
- Function templates are useful to reuse source code for functions. The given function may need to be rewritten for another data type.

```
int abs(int n) { // absolute value of ints
  return (n<0) ? -n : n; // if n is negative, return -n
}
long abs(long n){ // absolute value of longs
   return (n<0) ? -n : n;
}
float abs(float n){ // absolute value of floats
   return (n<0) ? -n : n;
}
```

Function templates

- In C++ overloading is supported; these functions can be written with the same name
- In C, each should have a different name: **abs(), fabs(), fabsl(), labs(), cabs(),** and so on.
- Rewriting wastes time and space.
- Furthermore, it is open to errors

- Function templates are used for such cases

Function templates, con't

```
// template used for absolute value function
template <class T>     // function template
T abs(T n){
     return (n < 0) ? -n : n;
}
int main(){
  int int1 = 5, int2 = -6;
  long lon1 = 70000L, lon2 = -80000L;
  double doub1 = 9.95, doub2 = -10.15;
  // calls instantiate functions
  cout << abs(int1) << endl;          // abs(int)
  cout << abs(int2) << endl;          // abs(int)
  cout << abs(lon1) << endl;          // abs(long)
  cout << abs(lon2) << endl;          // abs(long)
  cout << abs(doub1) << endl;         // abs(double)
  cout << abs(doub2) << endl;         // abs(double)
 ....
```

Function templates - components

- The template keyword signals the compiler that a function template is define.
- The keyword class, within the angle brackets, might just as well be called type.
- The variable following the keyword class (T in this example) is called the *template argument*.

Function template code generation

- The function template itself doesn't cause the compiler to generate any code. It can't generate code because it doesn't know yet what data type the function will be working with. It simply remembers the template for possible future use.
- Code is generated (compiled) according to the function call statement. This happens in expressions such as **abs(int1)** in the statement:
  - `cout << abs(int1);`

Function template code generation, con't

- When the compiler sees a function call, it knows that the type to use is **int**.
- So it generates a specific version of the abs(**T** n) function for type int, substituting **int** wherever it sees the name **T** in the function template.
- **int** → **T**
- This is called *instantiating* the function template, and each instantiated version of the function is called a *template function*.

See Example: e11_1.cpp

Template functions - constraints

- Data type used in the template function must support operations performed in the function template. For example in the abs function two operators are used: (n < 0) and -n .
- Each data type, which supports these operators can be used with the abs function.

Function templates - advantages

- Notice that the amount of RAM used by the program is the same whether function templates are used or there are separate functions.
- But, function templates make the listing shorter and easier to understand.
- Also, if the way the function works is needed to be changed, only one change in the function listing is enough.

Objects as template arguments

- A template function MAX can find maximum of two integers, floating point numbers or complex numbers. Integers and floats are built-in types, complex is a user defined type (class).

```
class ComplexT{ // A class to define complex numbers
    float re, im;
  public:
           :   // other member functions
    bool operator>(const ComplexT&) const ;   // header of
   operator> function
};
/* The Body of the function for operator > */
bool ComplexT::operator>(const ComplexT& z) const {
   float f1 = re * re + im * im;
    float f2 = z.re * z.re + z.im * z.im;
    return f1 > f2;
}
```

Objects as template arguments, con't

```
// template function
template <class type>
const type & MAX(const type &v1, const type & v2){
    if (v1 > v2) return v1;
       else return v2;
}
int main(){
   int i1=5,  i2= -3;
   char c1='D',  c2='N';
   float f1=3.05,  f2=12.47;
   ComplexT z1(1.4, 0.6),  z2(4.6, -3.8);
   cout << MAX(i1,i2) << endl;
   cout << MAX(c1,c2) << endl;
   cout << MAX(f1,f2) << endl;
   cout << MAX(z1,z2) << endl;     // operator << must be
   overloaded to print ComplexT
   return 0;
}
```

See Example: e11_2.cpp

Function templates with multiple arguments

```
// function returns index number of item, or -1 if not
   found template
template <class atype>
int find(const atype* array, atype value, int size){
    for(int j = 0;  j < size; j++)
       if( array[ j ] == value ) return j;
    return -1;
}

char chrArr[ ] = {'a', 'c', 'f', 's', 'u', 'z'}; // array
char ch = 'f';                                   // value to find
int intArr[ ] = {1, 3, 5, 9, 11, 13};
int in = 6;
double dubArr[ ] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;
```

Template arguments must match

```
int main() {
    cout << "\n 'f' in chrArray: index=" << find(chrArr, ch, 6);
    cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
    cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
    return 0;
}
```

Here, template argument is `atype`

```
int intarray[ ] = {1, 3, 5, 7};     // int array
float f1 = 5.0;                     // float value
int value = find(intarray, f1, 4); // ERROR!
```

- because the compiler expects all instances of atype to be the same type. It can generate a function `find(int*, int, int);` but it can't generate `find(int*, float, int);`

More than one template argument

- More than one template argument may exists in a function template.
- For example, in the find() function template, type of the size of the array may be determined when the function call is made. If the array is too large, then type long would be necessary for the array size instead of type int.

```
template <class atype, class btype>
   btype find(const atype* array, atype value, btype size){
       for( btype j = 0;  j < size;  j++) //note use of btype
           if( array[j] == value ) return j;
       return static_cast<btype>(-1);

   }
```

More than one template argument , con't

```
short int result , si = 100;
int invalue = 5;
result = find(intArr, invalue, si)

long lonresult, li = 100000;
float fvalue = 5.2;
lonresult = find(floatArr, fvalue, li)
```

- Note that multiple template arguments can lead to many functions being instantiated from a single template.
- Two such arguments, if there were six basic types that could reasonably be used for each one, would allow the creation of up to 36 functions.
- This can take up a lot of memory if the functions are large. On the other hand, you don't instantiate a version of the function unless you actually call it.

Templates vs. Macros

- The abs() function could be defined as
- `#define abs(n) ( (n<0) ? (-n) : (n) )`
- But
  - Macros don't perform any type checking. There may be several arguments to the macro that should be of the same type, but the compiler won't check whether or not they are.
  - Also, the type of the value returned isn't specified so the compiler can't tell if you're assigning it to an incompatible variable.
  - In any case, macros are confined to functions that can be expressed in a single statement
  - On the whole, it's best to avoid them.

Class templates

- The template concept can be applied to classes as well as to functions.
- Class templates are generally used for data storage (container) classes. Stacks and linked lists, are examples of data storage classes.
- The previous examples of the classes could store data of only a single basic type.
- Class templates can store different data types

Class templates - example

```
class Stack {
        int st[MAX];              // array of ints
        int top;                  // index number of top of stack
      public:
        Stack();                  // constructor
        void push(int var);       // takes int as argument
        int pop();                // returns int value
    };
```

The same class which stores data of type long:

```
    class LongStack {
        long st[MAX];             // array of longs
        int top;                  // index number of top of stack
      public:
        LongStack();              // constructor
        void push(long var);      // takes long as argument
        long pop();               // returns long value
     };
```

Class templates – example, con't

- Solution with a class template:

```
template <class Type>
class Stack{
      enum {MAX=100};
      Type st[MAX];              // stack: array of any type
      int top;                   // number of elements in the stack
   public:
      Stack(){top = 0;}          // constructor
      void push(Type);           // put number on stack
      Type pop();                // take number off stack
};

template<class Type>
void Stack<Type>::push(Type var){ // put number on stack
   if(top > MAX-1)               // if stack full,
     throw "Stack is full!";     // throw exception
   st[top++] = var;
}
```

Class templates – example, con't

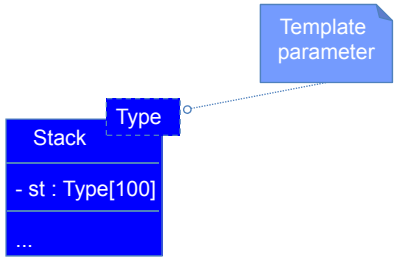
```
template<class Type>
Type Stack<Type>::pop(){  // take number off stack
   if(top <= 0)               // if stack empty,
        throw "Stack is empty!";          // throw exception
   return st[--top];
}
int main(){
   // s1 is object of class Stack<float>
   Stack<float> s1;
   // push 2 floats, pop 2 floats
   try{
     s1.push(1111.1);
     s1.push(2222.2);
     cout << "1: " << s1.pop() << endl;
     cout << "2: " << s1.pop() << endl;
   }
   // exception handler
   catch(const char * msg) {
      cout << msg << endl;
   }
```

Class templates – example, con't

```
// int main(){ continues...
// s2 is object of class Stack<long>
   Stack<long> s2;
   // push 2 longs, pop 2 longs
   try{
     s2.push(123123123L);
     s2.push(234234234L);
     cout << "1: " << s2.pop() << endl;
     cout << "2: " << s2.pop() << endl;
   }
   // exception handler
   catch(const char * msg) {
      cout << msg << endl;
   }
   return 0;
}      // End of program
```

See Example: e11_3.cpp

UML Notation for Template Classes



An object of template Stack.
In this example intStack object is an integer Stack .