Templates

Templates

- Type-independent patterns that can work with multiple data types.
  - Generic programming
  - Code reusable
- Function Templates
  - These define logic behind the algorithms that work for multiple data types.
- Class Templates
  - These define generic class patterns into which specific data types can be plugged in to produce new classes.

Function and function templates

- C++ routines work on specific types. We often need to write different routines to perform the same operation on different data types.

```
int maximum(int a, int b, int c)
 {
     int max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

Function and function templates

```
float maximum(float a, float b, float c)
 {
      float max = a;
      if (b > max) max = b;
      if (c > max) max = c;
      return max;
}
```

Function and function templates

```
double maximum(double a, double b, double c)
 {
      double max = a;
      if (b > max) max = b;
      if (c > max) max = c;
      return max;
}
```

The logic is exactly the same, but the data type is different.

Function templates allow the logic to be written once and used for all data types – generic function.

Function Templates

- Generic function to find a maximum value (see maximum example).

```
Template <class T>
T maximum(T a, T b, T c)
 {
     T max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

- Template function itself is incomplete because the compiler will need to know the actual type to generate code. So template program are often placed in .h files to be included in program that uses the function.
- C++ compiler will then generate the real function based on the use of the function template.

Function Templates Usage

- After a function template is included (or defined), the function can be used by passing parameters of real types.

```
Template <class T>
T maximum(T a, T b, T c)
…
int i1, i2, i3;
…
Int m = maximum(i1, i2, i3);
```

- maximum(i1, i2, i3) will invoke the template function with T==int. The function returns a value of int type.

Function Templates Usage

- Each call to `maximum()` on a different data type forces the compiler to generate a different function using the template. See the maximum example.
  - One copy of code for many types.

```
int i1, i2, i3;
// invoke int version of maximum
cout << "The maximum integer value is: “  << maximum( i1, i2, i3 );

// demonstrate maximum with double values
double d1, d2, d3;
// invoke double version of maximum
cout << "The maximum double value is: “  << maximum( d1, d2, d3 );
```

Another example

```
template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";
      cout << endl;
}
```

Usage

```
template< class T >
void printArray( const T *array, const int count );

char  cc[100];
int      ii[100];
double dd[100];
……
printArray(cc, 100);
printArray(ii, 100);
printArray(dd, 100);
```

Usage

```
template< class T >
void printArray( const T *array, const int count );

char  cc[100];
int      ii[100];
double dd[100];
myclass xx[100];  <- user defined type can also be used.
……
printArray(cc, 100);
printArray(ii, 100);
printArray(dd, 100);
printArray(xx, 100);
```

Use of template function

- Can any user defined type be used with a template function?
  - Not always, only the ones that support all operations used in the function.
  - E.g. if myclass does not have overloaded << operator, the printarray template function will not work.

Class Templates

- Class templates
  - Allow type-specific versions of generic classes
- Format:

```
template <class T>
class ClassName{
 definition
 }
```

  - Need not use **"T"**, any identifier will work
  - To create an object of the class, type

    ClassName`< type > myObject;`

    Example: `Stack< double > doubleStack;`

Class Templates

- Template class functions
  - Declared normally, but preceded by **`template<class T>`**
    - Generic data in class listed as type **`T`**
  - Binary scope resolution operator used
  - Template class function definition:

```
template<class T>
MyClass< T >::MyClass(int size)
{
   myArray = new T[size];
}
```

    - Constructor definition - creates an array of type **`T`**

```
1  // Fig. 22.3: tstack1.h
2  // Class template Stack
3  #ifndef TSTACK1_H
4  #define TSTACK1_H
5
6  template< class T >
7  class Stack {
8  public:
9     Stack( int = 10 );    // default constructor (stack size 10)
10    ~Stack() { delete [] stackPtr; } // destructor
11    bool push( const T& ); // push an element onto the stack
12    bool pop( T& );        // pop an element off the stack
13 private:
14    int size;             // # of elements in the stack
15    int top;              // location of the top element
16    T *stackPtr;          // pointer to the stack
17
18    bool isEmpty() const { return top == -1; }      // utility
19    bool isFull() const { return top == size - 1; } // functions
20 };
21
22 // Constructor with default size 10
23 template< class T >
24 Stack< T >::Stack( int s )
25 {
26    size = s > 0 ? s : 10;
27    top = -1;                // Stack is initially empty
28    stackPtr = new T[ size ]; // allocate space for elements
29 }
```

```
30
31 // Push an element onto the stack
32 // return 1 if successful, 0 otherwise
33 template< class T >
34 bool Stack< T >::push( const T &pushValue )
35 {
36    if ( !isFull() ) {
37       stackPtr[ ++top ] = pushValue; // place item in Stack
38       return true;  // push successful
39    }
40    return false;     // push unsuccessful
41 }
42
43 // Pop an element off the stack
44 template< class T >
45 bool Stack< T >::pop( T &popValue )
46 {
47    if ( !isEmpty() ) {
48       popValue = stackPtr[ top-- ];  // remove item from Stack
49       return true;  // pop successful
50    }
51    return false;     // pop unsuccessful
52 }
53
54 #endif
```

```
55 // Fig. 22.3: fig22_03.cpp
56 // Test driver for Stack template
57 #include <iostream>
58
59 using std::cout;
60 using std::cin;
61 using std::endl;
62
63 #include "tstack1.h"
64
65 int main()
66 {
67    Stack< double > doubleStack( 5 );
68    double f = 1.1;
69    cout << "Pushing elements onto doubleStack\n";
70
71    while ( doubleStack.push( f ) ) { // success true returned
72       cout << f << ' ';
73       f += 1.1;
74    }
75
76    cout << "\nStack is full. Cannot push " << f
77         << "\n\nPopping elements from doubleStack\n";
78
79    while ( doubleStack.pop( f ) )  // success true returned
```

```
80       cout << f << ' ';
81
82    cout << "\nStack is empty. Cannot pop\n";
83
84    Stack< int > intStack;
85    int i = 1;
86    cout << "\nPushing elements onto intStack\n";
87
88    while ( intStack.push( i ) ) { // success true returned
89       cout << i << ' ';
90       ++i;
91    }
92
93    cout << "\nStack is full. Cannot push " << i
94         << "\n\nPopping elements from intStack\n";
95
96    while ( intStack.pop( i ) )  // success true returned
97       cout << i << ' ';
98
99    cout << "\nStack is empty. Cannot pop\n";
100   return 0;
101}
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

- Program Output

Templates and Inheritance

- A class template can be derived from a template class
- A class template can be derived from a non-template class
- A template class can be derived from a class template
- A non-template class can be derived from a class template

Templates and friends

- Friendships allowed between a class template and
  - Global function
  - Member function of another class
  - Entire class
- **`friend`** functions
  - Inside definition of class template **`X`**:
  - **`friend void f1();`**
    - **`f1()`** a **`friend`** of all template classes
  - **`friend void f2( X< T > & );`**
    - **`f2( X< int > & )`** is a **`friend`** of **`X< int >`** only. The same applies for **`float`**, **`double`**, etc.
  - **`friend void A::f3();`**
    - Member function **`f3`** of class **`A`** is a **`friend`** of all template classes

Templates and friends

- `friend void C< T >::f4( X< T > & );`
  - `C<float>::f4( X< float> & )` is a `friend` of `class X<float>` only

- **`friend`** classes
  - `friend class Y;`
    - Every member function of `Y` a friend with every template class made from `X`
  - `friend class Z<T>;`
    - Class `Z<float>` a `friend` of class `X<float>`, etc.

Another Class Template Example

- MemoryCell template can be used for any type Object.

- Assumptions
  - Object has a zero parameter constructor
  - Object has a copy constructor
  - Copy-assignment operator

- Convention
  - Class templates declaration and implementation usually combined in a single file.
  - It is not easy to separate them in independent files due to complex c++ syntax.
  - This is different from the convention of separating class interface and implementation in different files.

```
1   /**
2    * A class for simulating a memory cell.
3    */
4   template <typename Object>
5   class MemoryCell
6   {
7     public:
8       explicit MemoryCell( const Object & initialValue = Object( ) )
9         : storedValue( initialValue ) { }
10      const Object & read( ) const
11        { return storedValue; }
12      void write( const Object & x )
13        { storedValue = x; }
14    private:
15      Object storedValue;
16  };
```

Class Template Usage Example

- MemoryCell can be used to store both primitive and class types.
- Remember
  - `MemoryCell` is not a class.
  - It's a class template.
  - `MemoryCell<int>`, `MemoryCell<string>` etc are classes.

```
1   int main( )
2   {
3       MemoryCell<int>    m1;
4       MemoryCell<string> m2;( "hello" );
5
6       m1.write( 37 );
7       m2.write( m2.read( ) + "world" );
8       cout << m1.read( ) << endl << m2.read( ) << endl;
9
10      return 0;
11  }
```