UEE1303
Objective-Oriented Programming

C++_Lecture 08:

Templates –

Function Templates and Class Templates

C: How to Program 8th ed.

Agenda

- Usefulness of Function Templates
  - create function templates
  - use multiple parameters in function templates
  - overload function templates (Chapter 23.6)
- Usefulness of Class Templates (Chapter 23.2)
  - Template parameters
    - type parameters and non-type parameters (Chapter 23.4-23.5)
  - create a complete class template
- Friend and inheritance in templates

Function Templates

- Most high-level languages require each function to have its own name
  - Can lead to a profusion of names
- Example: functions to find the absolute value
  - Three separate functions and prototypes are required

```
void abs(int);
void fabs(float);
void dabs(double);
```

- Each function performs the same operation
  - Only difference is the data type in argument

Example of Function Template

```
template <class T>
void showabs(T number)
{
    if (number < 0)
        number = -number;
    cout << "The absolute value of the number "
         << " is " << number << endl;
    return;
}
```

- Template allows for one function instead of three
  - T represents a general data type
  - T is replaced by an actual data type when compiler encounters a function call

Example of Function Template (cont.)

```
int main()
{
    int num1 = -4;
    float num2 = -4.23F;
    double num3 = -4.23456;
    showabs(num1);
    showabs(num2);
    showabs(num3);
    return 0;
}
```

- screen output

```
The absolute value of the number is 4
The absolute value of the number is 4.23
The absolute value of the number is 4.23456
```

Overloading Functions

- C++ allows multiple overloading functions
  - but need to define individually

```
void swap(int &r1, int& r2) {
    int tmp = r1;
    r1 = r2;
    r2 = tmp;                   }
void swap(long &r1, long& r2) {
    long tmp = r1;
    r1 = r2;
    r2 = tmp;                     }
void swap(double &r1, double& r2) {
    double tmp = r1;
    r1 = r2;
    r2 = tmp;                         }
```

Usefulness of Function Templates

- Ideally, you could create just one function with a variable name standing in for the type
  - Good idea but doesn't quite work in C++

```
void swap(varType& t1, varType& t2) {
    varType tmp = t1;
    t1 = t2;
    t2 = tmp;
}
```

- You need to create a template definition
  - Similar with some extra thing

Function Templates

- Using function template to simplify

```cpp
template <class T>
void swap(T& t1, T& t2) {
    T tmp = t1;
    t1 = t2;
    t2 = tmp;
}
```

instantiation

```cpp
void swap(
    int& t1,
    int& t2)
{...}
```

```cpp
void swap(
    long& t1,
    long& t2)
{...}
```

```cpp
void swap(
    double& t1,
    double& t2)
{...}
```

Creating Function Templates

- Function templates: *functions that use variable types*
  - outline for a group of functions that differ in datatypes of parameters they use
- A group of functions that generates from the same template is often called a family of functions
- In a function template, *at least one argument is generic* (or *parameterized*)
- You write a function template and it generates one or more template functions

Creating Function Templates (cont.)

```
template <class T>
void swap(T& t1, T& t2) {
    T tmp = t1;
    t1 = t2;
    t2 = tmp;
}
```

- Using the keyword `class` in the template definition does not necessarily mean than `T` stands for a *programmer-created class* type
- Many newer C++ compilers allow you to replace `class` with `typename` in the template definition

Creating Function Templates (cont.)

- When calling a function template, compiler determines type of actual argument passed

```
double a = 5, b = 3;
swap(a, b);
```

  - designating parameterized type is *implicit*

- The compiler *generates* code for different functions as it needs
  - depend on the function calls

```
void swap(double& t1, double& t2) {
    double tmp = t1;
    t1 = t2;
    t2 = tmp;
}
```

Function Template printArray

```
// Fig. 22.01: fig22_01.cpp (7th)
// Using template functions
#include <iostream>
using namespace std;
// function template printArray definition
template <typename T>
void printArray (const T *array, int count)
{
    for ( int i = 0, i < count; i++)
        cout << array[i] << " ";
    cout << endl;
}
int main ()
{
    const int aCount = 5;
    const int bCount = 7;
    const int cCount = 6;
```

Function Template printArray (cont.)

```
    int a[aCount] = {1, 2, 3, 4, 5};
    double b[bCount] = {1.1, 2,2, 3.3, 4.4, 5.5, 6.6, 7.7};
    char c[cCount] = "HELLO"; // 6th position for null
    cout << "Array a contains: " << endl;
    printArray( a, aCount);

    cout << "Array b contains: " << endl;
    printArray( b, bCount);

    cout << "Array c contains: " << endl;
    printArray( c, cCount);
}
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Using Multiple Parameters

- x, y, z, and max may be of any type for which the > operator and the = operator have been defined
- x, y, z, and max all must be of the same type because they are all defined to be the same type, named T

```
template <class T>
T FindMax(
    T x, T y, T z) {
    T max = x;
    if (y > max)
        max = y;
    if (z > max)
        max = z;
    return max;
}
```

Using Multiple Parameters (cont.)

```
class PhoneCall
{
private:
    int minutes;
public:
    PhoneCall(int min=0) {minutes = min;}
    bool operator>(PhoneCall&);
    friend ostream& operator<<(ostream&,
        PhoneCall);
};
```

Using Multiple Parameters (cont.)

```
bool PhoneCall::operator>(PhoneCall& call)
{
    bool isTrue = false;
    if (minutes > call.minutes)
        isTrue = true;
    return isTrue;
}
ostream& operator<<(ostream& out,
                    PhoneCall call)
{
    out << "Phone call that last "
        << call.minutes
        << " minutes\n";
    return out;
}
```

Using Multiple Parameters (cont.)

```
//in main()
    int a; double b;
    PhoneCall c1(4), c2(6), c3(11), c(0);
    a = FindMax(3, 5, 4);
    b = FindMax(12.3, 5.9, 25.4);
    c = FindMax(c1, c2, c3);
    cout << a << "\n";
    cout << b << "\n";
    cout << c << "\n";
```

- screen output

```
5
25.4
Phone call that last 11 minutes
```

Overloading Function Templates

- Can overload function templates only when each version takes a different argument list
  - allow compiler to distinguish

```
template <class T>
T FindMax(T x, T y) {
    T max = x;
    if (y > max)
        max = y;
    return max;
}
```

```
template <class T>
T FindMax(
    T x, T y, T z) {
    T max = x;
    if (y > max)
        max = y;
    if (z > max)
        max = z;
    return max;
}
```

Overloading Function Templates (cont.)

```
//in main()
    int x1 = 1, x2 = 2, x3 = 3, x;
    double y1 = 3.3, y2 = 2.2, y3 = 1.1, y;
    //call FindMax with 2 and 3 integers
    cout << FindMax(x1, x2) << " versus "
         << FindMax(x1, x2, x3) << “\n”;
    //call FindMax with 2 and 3 doubles
    cout << FindMax(y1, y2) << " versus "
         << FindMax(y1, y2, y3) << “\n”;
```

```
2 versus 3
3.3 versus 3.3
```

More than One Type

```
template <class T>
void repeatValue(T val, int times) {
    for (int x = 0; x < times; ++x) {
        cout<<"#"<<(x+1)<<" "<<val<<"\n"; }
}
class Store {
    int  storeid;
    string address;
    string manager;
public:
    Store(int sid, string add, string mgr){
         storeid = sid;
         address = add;
         manager = mgr;                        }
    friend ostream& operator<<(ostream&,
         Store);
};
```

More than One Type (cont.)

```
ostream& operator<<(ostream& out,
                    Store store) {
    out << "Str:" << store.storeid << " "
        << "Add:" << store.address << " "
        << "Mgr:" << store.manager ;
    return out;                          }
```

```
//in main()
    double a = 3.0; char b = 'B';
    string c = "good";
    Store d(113, "23 Ave. Q", "Jacky");
    repeatValue(a, 3);
    repeatValue(b, 2);
    repeatValue(c, 4);
    repeatValue(d, 2);
```

More than One Type (cont.)

- screen output

```
#1 3.0
#2 3.0
#3 3.0
#1 B
#2 B
#1 good
#2 good
#3 good
#4 good
#1 Str: 113 Add: 23 Ave. Q Mgr: Jacky
#2 Str: 113 Add: 23 Ave. Q Mgr: Jacky
```

More than One Parameterized Type

```
template <class T, class U>
void ShowCompare(T v1, U v2) {
    if (v1 == v2)     // redefine ==
        cout << "v1 is equal to v2\n";
    else if (v1 > v2) // redefine >
        cout << "v1 is bigger than v2\n ";
    else //if (v1 << v2)
        cout << "v1 is smaller than v2\n";
}
```

```
class PhoneCall {
    int minutes;
public:
    PhoneCall(int min=0) {minutes = min;}
    friend ostream& operator<<(ostream&,
        PhoneCall); //same as prog in p.16
```

More than One Parameterized Type (cont.)

```
    //overloading operator >
    bool operator>(PhoneCall c) {
        return (minutes>c. minutes)? true:
false;  }
    bool operator>(int min) {
        return (minutes>min)? true:
false;     }
    //overlaoding operator==
    bool operator==(PhoneCall c) {
        return (minutes==c.minutes)? true:
false; }
    bool operator==(int min) {
        return (minutes==min)? true:
false;    }

};
```

More than One Parameterized Type (cont.)

```
//in main()
    int a = 68; double b = 68.5; char c='D';
    PhoneCall d(3), e(5);
    ShowCompare(a,68); ShowCompare(a,b);
    ShowCompare(a,c);  ShowCompare(d,a);
    ShowCompare(d,e);  ShowCompare(d,3);
```

- screen output

```
v1 is equal to v2
v1 is smaller than v2
v1 is equal to v2
v1 is smaller than v2
v1 is smaller than v2
v1 is equal to v2
```

Specifying Type Explicitly

- When calling a template function, the arguments dictate the types to be used
- To override a deduced type:

```
someFunction<char>(someArgument)
```

  - useful when at least one of the types you need to generate in the function is not an argument
- Example:

```
template <class T>
T doubleValue (T val) {
    val *=2;
    return val;
}
```

Specifying Type Explicitly (cont.)

```
//in main()
    int a = 6; double b = 7.4;
    cout <<a<<" & "<<doubleVal(a)<<"\n";
    cout <<b<<" & "<<doubleVal(b)<<"\n";
    cout <<b<<" & "<<doubleVal<int>(b)
         <<"\n";
```

- screen output

```
6 & 12
7.4 & 14.8
7.4 & 14
```

Specifying Multiple Types Explicitly

- To override multiple deduced types:

```
someFunction<type1, type2, …>
                  (arg1, arg2, …);
```

- Example:

```
template <class T, class U>
T tripleValue (U val) {
    T tmp = val*3;
    return tmp;
}
```

Specifying Multiple Types Explicitly (cont.)

```
//in main()
    int a = 4; double b = 8.8;
    cout << tripleVal<int>(a) << "\n";
    cout << tripleVal<int>(b) << "\n";
    cout << tripleVal<int,double>(b)<< "\n";
    cout << tripleVal<int, int>(b) <<"\n";
```

- screen output

```
12
26
26
24
```

Class Templates

- A class template defines a family of classes
  - serve as *class outline* to generate many classes
  - specific classes are generated during *compile time*
- Class templates promote code reusability
  - reduce program development time
  - used for a need to create several similar class ⇨ at least one type is generic (parameterized)
- Terms "class template" and "template class" are used interchangeably

Example for Class Templates

- Example: you may want to use

```
Number<int> myValue(25);
Number<double> yourValue(3.46);
```

```
template <class T>
class Number {
    T number;
public:
    Number(T num) { number = num; }
    void ShowNumber() {
        cout << "Number = "
             << number << endl;
    }
};
```

Example for Class Templates (cont.)

```
//in main()
    Number<int> a(65); a.ShowNumber();
    Number<double> b(8.8); b.ShowNumber();
    Number<char> c('D'); c.ShowNumber();
    Number<int> d('D'); d.ShowNumber();
    Number<char> e(70); e.ShowNumber();
```

```
Number = 65
Number = 8.8
Number = D
Number = 68
Number = F
```

Template Parameters

- 3 forms of template parameters
  - type parameters
  - non-type parameters
  - template parameters with default arguments
- A type parameter defines a type identifier
  - when instantiating a template class, a specific datatype listed in the argument list substitute for the type identifier
  - either `class` or `typename` must precede a template type parameter

Example of Type Parameters

- Example:

```
template <class C1, class C2, class C3>
class X { //... };
```

```
template <typename T1, typename T2>
class Y { //... };
```

  - type identifies: `C1, C2, C3, T1, T2`

- Substitute type identifiers with specific datatypes when instantiating objects

```
//C1=int, C2=double, C3=int
X<int, double, int> p;
Y<char, int> q;    //T1=char, T2=int
Y<int, double*> r; //T1=int, T2=double*
```

Non-Type Parameters

- A non-type parameter can be
  - Integral types: `int`, `char` and `bool`
  - enumeration type
  - reference to object or function
  - pointer to object, function or member
- A non-type parameter cannot be
  - floating types: `float` and `double`
  - user-defined class type
  - type `void`

Example of Non-type Parameters

- Good examples:

```
template <int A, char B, bool C>
class G1 { //... };
```

```
template <float* D, double& E>
class G2 { //... };
```

- Bad example:

```
template <double F>
class B1 { //... }; // cannot be double
```

```
template <PhoneCall P>
class B2 { //... };  //cannot be class
```

More on Non-Type Parameters

- A template parameter may have a default argument

```
template <class T=int, int n=10>
class C3 { //... };
```

  - one or both argument can be optional

```
C3 < > a;
C3 b; // error: missing < >
C3 <double, 50> c;
C3 <char> d;
C3 <20> e; // error: missing template
           // argument
```

Example of Stack Template

```
// Fig. 22.2: Stack.h (7th)
#ifndef STACK_H
#define STACK_H
template <typename T>
class Stack
{
public:
    Stack( int = 10); // default constructor (size = 10)
    ~Stack() { delete [] stackPtr; }
    bool push ( const T & );
    bool pop (T &);
    bool isEmpty() const { return top == -1; }
    bool isFull() const { return top == size -1; }
private:
    int size;   // # of elements in the Stack
    int top;    // location of the top element (-1: empty)
    T *stackPtr;
}; // end class Time
```

Example of Stack Template (cont.)

```
template <typename T>
Stack<T>::Stack( int s ):size( s > 0? s : 10),
                         top(-1),
                         stackPtr( new T[size] )
{ // empty body }
template <typename T>
bool Stack<T>::push( const T &pushValue)
{
    if ( !isFull() )
    {
        stackPtr [++top] = pushValue; // place item
        return true;
    }
    return false;
}
```

Example of Stack Template (cont.)

```
template <typename T>
bool Stack<T>::pop( T &popValue)
{
    if ( !isEmpty() )
    {
        popValue = stackPtr [ top-- ]; // remove item
        return true;
    }
    return false;
}
#endif
```

Example of Stack Template (cont.)

```
// Fig. 22.3: fig22_03.cpp (7th)
#include <iostream>
#include "Stack.h"
using namespace std;
int main()
{
    Stack< double > doubleStack(5); // size 5
    double doubleValue = 1.1;
    cout << "Pushing elements onto doubleStack\n";

    // push 5 double onto doubleStack
    while ( doubleStack.push( doubleValue ))
    {
        cout << doubleValue << ' ';
        doubleValue += 1.1;
    }
    cout << "\Stack is full. Cannot push " << doubleValue
         << "\n\nPopping elements from doubleStack\n";
```

Example of Stack Template (cont.)

```
    // pop elements from doubleStack
    while ( doubleStack.pop( doubleValue ))
        cout << doubleValue << ' ';
    cout << "\Stack is empty. Cannot pop\n";

    Stack< int > intStack;
    int intValue = 1;
    cout << "Pushing elements onto intStack\n";
    // push 10 int onto intStack
    while ( intStack.push( intValue ))
        cout << intValue++ << ' ';
    cout << "\Stack is full. Cannot push " << intValue
         << "\n\nPopping elements from intStack\n";
    // pop elements from intStack
    while ( intStack.pop( intValue ))
        cout << intValue << ' ';
    cout << "\Stack is empty. Cannot pop\n";
}
```

Example of Stack Template (cont.)

- screen output

```
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
```

Example of Stack Template (cont.)

```
// Fig. 22.4: fig22_04.cpp (7th)
#include <iostream>
#include <string>
#include "Stack.h"
using namespace std;

template<typename T>
void testStack(
    Stack <T> &theStack, // reference to Stack<T>
    T value, // initial value to push
    T increment, // increment for subsequent values
    const string stackName) // name of the Stack<T> object
{
    cout << "\nPushing elements onto " << stackName << '\n';
    // push element onto Stack
    while ( theStack.push( value )) {
        cout << value << ' ';
        value += increment;
    }
```

Example of Stack Template (cont.)

```
    cout << "\nStack is full. Cannot push " << value
         << "\n\nPopping elements from " << stackName
         << '\n';
    // pop element from Stack
    while ( theStack.pop( value ))
        cout << value << ' ';
    cout << "\n Stack is emepty. Cannot pop" << endl;
}
int main()
{
    Stack<double> doubleStack(5); // size 5
    Stack<int> intStack;          // default size 10
    testStack(doubleStack, 1.1, 1.1, "doubleStack");
    testStack(intStack, 1, 1, "intStack");
}
```

Another Stack Template

```
template <class T, int MAXSIZE>
class CStack {
    T elems[MAXSIZE];
    int top;
public:
    CStack() { top = 0;}
    void push(T e) {
        if ( top == MAXSIZE ) {
            cout << "full"; return; }
        elems[top++] = e;      }
    T pop() {
        if ( top <= 0 ) {
            cout << "empty"; return; }
        top--;
        return elems[top];            }
    bool empty() { return (top == 0);   }
    bool full() { return (top == MAXSIZE);}
};
```

Another Stack Template (cont.)

- If instantiating an object from `CStack` template

```
//in main()
    CStack<int, 25> CS;
```

- Compiler replace `T` with `int`, `MAXSIZE` with `25`

```
class CStack {
    int elems[25];
    int top;
public:
    CStack() { top = 0;}
    void push(int e) { //…                  }
    int  pop() { //…                        }
    bool empty() { return (top == 0);        }
    bool full() { return (top == 25);  }
};
```

Another Stack Template (cont.)

- Write a display function for `CStack` template

```
template <class T, int MAXSIZE>
void ShowStack(CStack<T, MAXSIZE> &s) {
    while (!s.empty())
        cout << s.pop();
    cout << endl;
}
```

```
//in main()
    CStack<int, 25> cs1;
    for (int i = 1; i < 10; ++i)
        cs1.push(i);
    ShowStack(cs1);
    CStack<char, 10> cs2;
    for (int j = 65; j < 70; ++j)
        cs1.push(j); // 65 is 'A'
    ShowStack(cs2);
```

Another Stack Template (cont.)

- screen output

```
9 8 7 6 5 4 3 2 1
E D C B A
```

Friends & Inheritance in Templates

- Friend functions can be used with template classes
  - same as with ordinary classes
  - simply requires proper type parameters
  - common to have friends of template classes, especially for operator overloading
- Nothing new for inheritance
- Derived template classes
  - can derive from template or non-template class
  - derived class is naturally a template class

Base Class Template

```
template <class T>
class TBase {
private:
    T x, y;
public:
    TBase() {}
    TBase(T a, T b) : x(a), y(b) {}
    ~TBase() {}
    T getX();
    T getY();
};
template <class T>
T TBase<T>::getX() { return x; }
template <class T>
T TBase<T>::getY() { return y; }
```

Derived Class & Class Template

- Derive non-class template from class template
  ⇨ easy to understand
  - behave like normal classes

```
class TDerived1: public TBase<int> {
private:
    int z;
public:
    TDerived1(int a, int b, int c):
        TBase<int>(a, b), z(c) {}
    int getZ() { return z; }
};
```

  - `TDerived1` is NOT a *class template*

Derived Class & Class Template (cont.)

- Derive *class template* from class template
  - same as the normal class inheritance

```
template <class T>
class TDerived2 : public TBase<T> {
private:
    T z;
public:
    TDerived2(T a, T b, T c):
        TBase<T>(a, b), z(c) {}
    T getZ() { return z; }
};
```

  - `TDerived2` is also a class template

Derived Class & Class Template (cont.)

- Derive *class template* from non-class template ⇨ process details carefully

```
template <class T>
class TDerived3 : public TDerived1 {
private:
    T w;
public:
    TDerived3(int a, int b, int c, T d):
    TDerived1(a, b, c), w(d) {}
    T getW() { return w; }
};
```

  - call `TDerived1` constructor with known datatypes for parameters

Main Program & Results

```
TBase<int> c1(0, 1);
cout << "TBase: x= " << c1.getX() << " y = " <<
c1.getY() << endl;
TDerived1 c2(1, 3, 5);
cout << "TDerived1: x = " << c2.getX() << " y = " <<
c2.getY() << " z = " << c2.getZ() << endl;
TDerived2<double> c3(2.2, 4.4, 6.6);
cout << "TDerived2: x= " << c3.getX() << " y = " <<
c3.getY() << " z = " << c3.getZ() << endl;
TDerived3<int> c4(3.5, 6.5, 9.5, 12.5);
cout << "TDerived3: x = " << c4.getX() << " y = " <<
c4.getY() << " z = " << c4.getZ() << " w=" <<
c4.getW() << endl;
```

```
TBase: x = 0 y = 1
TDerived1: x = 1 y = 3 z = 5
TDerived2: x = 2.2 y = 4.4 z = 6.6
TDerived3: x = 3 y = 6 z = 9 w = 12
```

Summary

- Tasks required by overloaded functions may be so similar that you create a lot of repetitious code
- Function templates serve as an outline or pattern for a group of functions that differ in the types of parameters they use
- Function templates can support multiple parameters
- You can overload function templates
  - Each takes a different argument list
- Function templates can use variables of multiple types

Summary (cont.)

- To create a function template that employs multiple generic types, you use a unique type identifier for each type
- When you call a template function, the arguments to the function dictate the types to be used
- You can use multiple explicit types when you call a template function
- If you need to create several similar classes, consider writing a template class (at least one type is generic or parameterized)

References

- Paul Deitel and Harvey Deitel, “C How to Program” Eighth Edition
  - Chapter 23
- Paul Deitel and Harvey Deitel, “C++ How to Program (late objects version)” Seventh Edition
  - Chapter 14
- W. Savitch, “Absolute C++,” Fourth Edition
  - Chapter 16