Computational Intelligence on Automation Lab @ NCTU

UEE1303(1070) S'12 Object-Oriented Programming in C++

Lecture 10: *Templates – Function Templates And Class Templates*

Learning Objectives (1/2)

- Learn about the usefulness of function templates
- Create function templates
- Use multiple parameters in function templates
- Overload function templates
- Create function templates with multiple data types
- Create function templates with multiple parameterized types

Learning Objectives (2/2)

- Explicitly specify the type in a function template
- Use multiple explicit types when you call a template function
- Learn about the usefulness of class templates
- Create a complete class template
- Learn about container classes
- Create an `Array` template class

Example for Templates

- 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

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

instantiation

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

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

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


Creating Function Templates (1/3)

- **Function templates:** *functions that use variable types*
  - outline for a group of functions that only differ in datatypes of parameters used
- 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 (2/3)

```
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 that **`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 (3/3)

- 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;
}
```


Using Multiple Parameters (1/4)

- Example:
- 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 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;
}
```

lec10-1.cpp

Using Multiple Parameters (2/4)

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

};
bool PhoneCall::operator>(PhoneCall& call){
    bool isTrue = false;
    if (minutes > call.minutes)
        isTrue = true;
    return isTrue;                          }
```

lec10-1.cpp

Using Multiple Parameters (3/4)

```
ostream& operator<<(ostream& out,
                    PhoneCall call) {
    out << “Phone call that lasts ”
        << call.minutes
        << “ minutes\n”;
    return out;                          }
```

lec10-1.cpp

```
//in main()
    int a; double b;
    PhoneCall c1(4), c2(6), c3(11), c;
    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”;
```

Using Multiple Parameters (4/4)

- Output of the program

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


Overloading Function Templates (1/2)

- 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;
    else if (z > max)
        max = z;
    return max;
}
```

lec10-2.cpp

Overloading Function Templates (2/2)

lec10-2.cpp

```
//in prog2 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 (1/3)

```
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);
};
```

lec10-3.cpp

More than One Type (2/3)

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

lec10-3.cpp

```
//in prog3 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 (3/3)

- Output of the program

```
>./prog3
#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 (1/3)

```
template <class T, class U>
void ShowCompare(T v1, U v2) {
    //cout << v1 << " versus " << v2 <<"\n";
    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";
}
```

lec10-4.cpp

```
class PhoneCall {
    int minutes;
public:
    PhoneCall(int min=0) {minutes = min;}
    friend ostream& operator<<(ostream&,
        PhoneCall); //same as prog1
```


More than One Parameterized Type (2/3)

lec10-4.cpp

```
//contunue class PhoneCall
    //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.minute)? true:
false;  }
    bool operator==(int min) {
        return (minutes==min)? true:
false;  }
```

More than One Parameterized Type (2/3)

lec10-4.cpp

```
//in prog4 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);
```

```
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 (1/2)

- 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:

lec10-5.cpp

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


Specifying Type Explicitly (2/2)

```
//in prog5 main()
    int a=6; double b=7.4;
    cout <<a<<“ & ”<<doubleVal(a)<<“\n”;
    cout <<b<<“ & ”<<doubleVal(b)<<“\n”;
    cout <<b<<“ & ”<<doubleVal<int>(b)
         <<“\n”;
```

lec10-5.cpp

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


Specifying Multiple Types Explicitly (1/2)

- To override multiple deduced types:
  ```
  someFunction<type1, type2, ...>
               (arg1, arg2, ...);
  ```
- Example:

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

lec10-6.cpp

Specifying Multiple Types Explicitly (2/2)

lec10-6.cpp

```
//in prog6 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";
```

```
12
26
26
24
```


Class Templates

- A class template defines a family of classes
  - serve as a *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 classes ⇨ at least one type is generic (parameterized)
- Terms "class template" and "template class" are used interchangeably

Example for Class Templates (1/2)

- 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 val) { number = val; }
  void ShowNumber() {
     cout << "Number = " << number << "\n";
  }
};
```

lec10-7.cpp

Example for Class Templates (2/2)

lec10-7.cpp

```
//in prog7 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*
- 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 `typname` must precede a template type parameter

Examples of Type Parameters

- Examples:

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

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

  - type identifiers: `C1`, `C2`, `C3`, `T1`, `T2`
- Substitute type identifiers with specific datatypes when instantiating objects

```
X<int, double, int> p; //C1,C3=int,C2=double
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`

Examples of Non-Type Parameters

- Good examples:

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

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

- Bad examples:

```
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
  - ex:

```
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 for Stack Template (1/4)

```
template <class T, int MAXSIZE>
class CStack {
     T elems[MAXSIZE];
     int top;
public:
     CStack() { top=0; }
     bool empty() { return (top==0); }
     bool full() { return (top==MAXSIZE); }
     void push(T e) {
         if (top==MAXSIZE) {
             cout << "full"; return; }
         elems[top++] = e;              }
     T pop() {
         if (top<=0){cout<<"empty";exit(-1);}
         top--; return elems[top];       }
};
```

lec10-8.cpp

Example for Stack Template (2/4)

- If instantiating an object from `CStack` template

```
//in prog8 main()
     CStack<int, 25> cs;
```

lec10-8.cpp

- Compiler replaces `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); }
};
```


Example for Stack Template (3/4)

- Write a display function for `CStack` template

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

lec10-8.cpp

```
//in prog8 main()
     CStack<int, 25> cs1;
     for (int idx=1; idx<10; ++idx)
          cs1.push(idx);
     ShowStack(cs1);
     CStack<char, 10> cs2;
     for (int jdx=65; jdx<70; ++jdx)
          cs2.push(jdx); //65 is 'A'
     ShowStack(cs2);
```

Example for Stack Template (4/4)

- Output of the program

```
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

lec10-9.cpp

```
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 (1/3)

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

lec10-9.cpp

```
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 (2/3)

- 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; }
};
```

lec10-9.cpp

– `TDerived2` is also a class template

Derived Class & Class Template (3/3)

- 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; }
};
```

lec10-9.cpp

– 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;
```

lec10-9.cpp

```
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 (1/2)

- 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 (2/2)

- 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 datatype is generic or parameterized)