Object-oriented programming (with C++)

Lecture #11: Templates and Friends

Outline

- Function Template
- Class Template
- Friends

Templates

- Templates give us the means to define a **family** of functions or classes that share the same functionality but which may differ with respect to the data type used internally
- A function template is a framework for generating related functions
- A class template is a framework for generating the source code for any number of related classes

Function Template

- A function can be defined in terms of an **unspecified** type
- The compiler generates separate versions of the function based on the type of the arguments passed in the function calls

Function Template

template <class T>

return-type function-name(T param)

// one parameter function

- T is called a template parameter

One Parameter Function Template

```
template <class T>
void display(const T &val) { cout << val; }
```

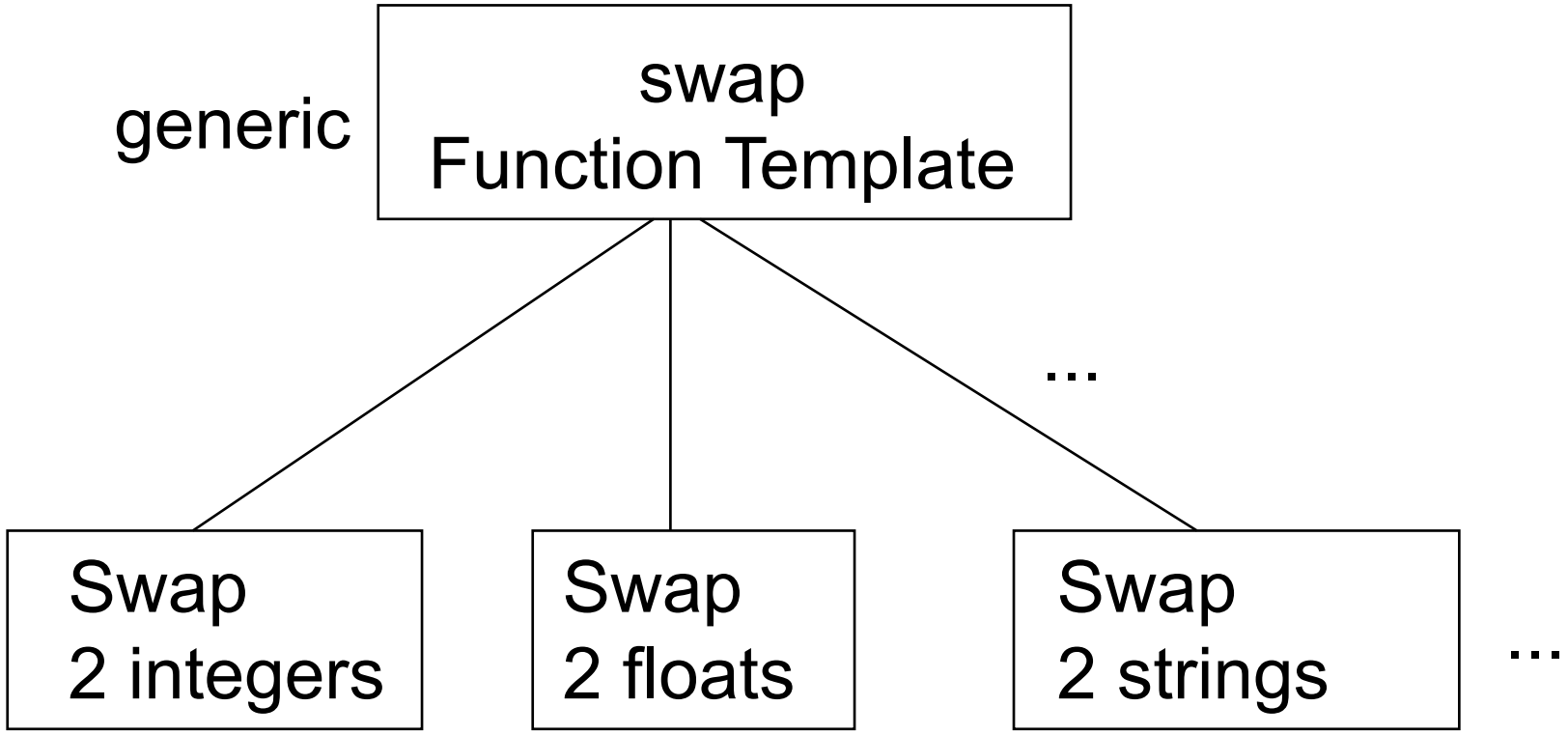
- One parameter function with an additional parameter which is note a template parameter:

```
template <class T>
void display(const T & val, osteam &os) { os << val; }
```

- The same parameter appears multiple times

```
template < class T>
void swap(T & x, T & y) {}
```

Swap


generic
swap
Function Template
...
Swap
2 integers
Swap
2 floats
Swap
2 strings
...

Swap

```
#include <iostream>
template <class T>
void swap(T & x, T & y) {
    T temp;
    temp = x;
    x = y;
    y = temp;
}
```

Swap

// Lec11_ex1-swap.cpp

Multiple Parameter Function Template

```
template <class T1, T2>
void arrayInput(T1   array, T2 & count) {
  for (T2 j= 0; j < count; j++) {
      cout << “value:”;
      cin >> array[j];
  }
}
```

Multiple Parameter Function Template

```
const unsigned tempCount = 3;
float temperature[tempCount];
const unsigned stationCount = 4;
int station[stationCount];
arrayInput(temperature, tempCount)
arrayInput(station, stationCount);
```

Table Lookup

```
template <class T>
long indexOf( T searchVal, const T * table, unsigned size )
{
  for (unsigned i = 0; i < size; i++)
    if (searchVal == table[i])
      return i;
  return -1;
}
```

Table Lookup

```
int main() {
  const unsigned iCount = 10, fCount = 5, sCount = 5;
  int iTable[iCount] = { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90 };
  foat fTable[fCount] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
  cout << indexOf( 20, iTable, iCount )   << endl;
  cout << indexOf( 2.2f, fTable, fCount ) << endl;
  string names[sCount] = { "John","Mary","Sue","Dan","Bob" };
  cout << indexOf( (string) "Dan", names, sCount ) << endl;
  return 0;
}
```

Note

- In a function template, if an operator is used, you must be sure every class relevant to the template will support the operator (such as “==”, etc.)
- We can use operator overloading to ensure compatibility

Student Comparison

```
class Student {
    public:
      Student( long idVal ) {  id = idVal; }
      int operator ==( const Student & s2 ) const {
        return id == s2.id;
      }
    private:
      long id;  // student ID number
};
```

Student Comparison

```
int main() {
    const unsigned sc= 5;
    Student sTable[sc] = { 10000, 11111, 20000, 22222, 30000 };
    Student s( 22222 );
    cout << indexOf( s, sTable, sCount ) << endl;     // print "3"
    return 0;
} //Lec11_ex3-student-comp.cpp
```

Overriding a Function Tempate

- When a function template does not apply to a particular type, it may be necessary to either
  - override the function template, or
  - make the type conform to the function template

Explicit Function Implemention

```
long indexOf( const char * searchVal, char * table[], unsigned size ) {
  for (unsigned i = 0; i < size; i++)
        if( strcmp(searchVal, table[i]) == 0 )
            return i;
  return -1;
}
```

Explicit Function Implementation

```
int main() {
    const unsigned iCount = 10, nCount = 5;
    int iTable[iCount] = { 0,10,20,30,40,50,60,70,80,90 };
    cout << indexOf( 20, iTable, iCount )  << endl;//2
    const char * names[nCount] =
      { "John","Mary","Sue","Dan","Bob" };
    cout << indexOf( "Dan", names, nCount ) << endl;//3
    return 0;
}//Lec11_ex4-expicit.cpp
```

Class Template

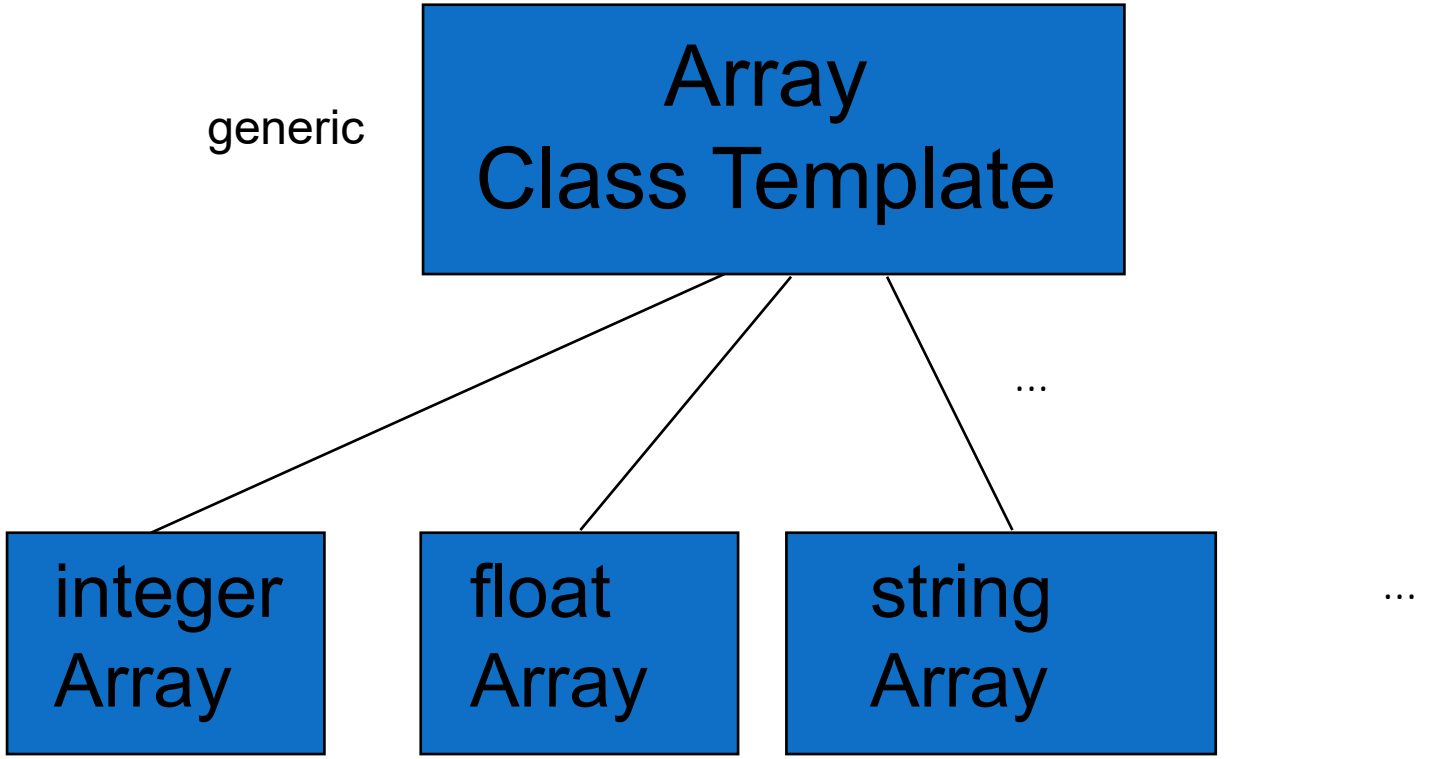
- Declare and define an object:

```
template <class T>
class MyClass {
// using T inside (parametrically)
};
MyClass <int> x;
MyClass <Student> a;
```

A Simple Example

```
template <class T1, class T2>
class Circle {
// ...
private:
  T1 x, y;
  T2 radius;
};
Circle <int, long> c1;
Circle <unsigned, float> c2;
```

Class Template



Array Class Template

```
template <class T>
class Array {
    public:
      Array( unsigned sz );
      ~Array();
      T & operator[]( unsigned i );
    private:
      T * values;
      unsigned size;
};
```

Array Class Template

```
template<class T> Array<T>::Array( unsigned sz ) {
  values = new T[sz]; size = sz;
}
template<class T> T & Array<T>::operator[] ( unsigned i ) {
  if( i >= size ) {
    cout << "ERROR: array index out of bound!!!\n”;  abort();
  }
  return values[i];
}
template<class T> Array<T>::~Array() {   delete [] values; }
```

Array Class Template

```
int main() {
  const unsigned numStudents = 2;
  Array<int>     ages( numStudents );
  Array<float>   gpas( numStudents );
  Array<string>  names(numStudents);
  for(int j = 0; j < numStudents; j++) {
        // do whatever you want
  }
  return 0;
} //Lec11_ex5-array.cpp
```

Templates and Inheritance

- A template is not a class
  - A template is not inherited
- A class based on a template is an ordinary class

```
class t1 : public Array<int> {...}//OK!!!
template <class T>
class Myclass : public aClass {
//OK!!!
};
```

Template and Static Members

- Remember
  - Template is not class
- Static members defined in a template are static members of the classes associated with a template

Friends

- Friend Classes
- Friend Functions

Friend Classes

- A friend class is a class in which all member functions have been granted full access to all the *(private, protected, and certainly public)* members of the class defining it as a friend (by instances)
- The friend class is declared inside the class granting friendship

Example

```
class C1 {
        friend class C2;
    //...
};
class C2{
        friend class C3;
    //...
};
```

Example

// Lec11_ex7-friend.cpp

Friend Functions

- A friend function is a function which has been granted full access to the private and protected members of an instance of the class
- The friend function is declared inside the class granting friendship

Example

//Lec11_ex8-friend-func.cpp

Properties of Friends

- Non-symmetrical
  - For example : If c2 is a friend of c1, but c1 is not necessarily a friend of c2.
- Non-transitive
  - For example: If c2 is a friend of c1 and c3 is a friend of c2, but c3 is not necessarily a friend of c1.

Properties of Friends

- Not inheritable
  - A friend of a base class is not inherited by derived classes

Example

```
class Employee {
      public:
        friend float calcPay(Employee &e);
      //..
};
class SalariedEmployee : public Employee{ //... };
```

Here, the function CalcPay() can not access the private members of SalariedEmployee!!!

Things to consider when using Friends

- Friends make loosely coupled classes tightly coupled, which may results in problems with modularity and error searching
- Friends diminish encapsulation and information hiding
- ***Limit the use of "friends"***