ECE 330 - Software Design

Module 2, Lecture 3: OO Design - OO Concepts in C++

Dr. Edward J Nava

University of New Mexico

UNM | Electrical & Computer Engineering

History of C++

- C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C language and originally named C with Classes. It was renamed C++ in 1983.
- Stroustrup was influenced by the concept of embedding abstract data types in a programming language, and in particular, the class construct found in the SIMULA 67.
- Stroustrup states that the C++ language was originally invented because he “wanted to write some event-driven simulations for which SIMULA 67 would have been ideal, except for efficiency considerations”, and that “the name signifies the evolutionary nature of the changes from C”.
- In 1998, an ISO committee standardized C++ (the published version of this informally became known as C++98).
- C++ is a statically typed, multi-paradigm, compiled, general-purpose, mid-level programming language.

THE UNIVERSITY of NEW MEXICO

Namespaces

In C/C++ the namespace keyword allows you to create a context for identifiers, meaning multiple identifiers in a C/C++ program can have the same name, as long as each of them is declared within a different namespace. Creating a namespace is a easy as:

```
namespace stuff {
        //identifiers go here
        }
```

and then identifiers within that namespace can be scoped using:

```
stuff::my_indentifier
```

An entire namespace can be introduced into a section of code using:

```
using namespace stuff;
```

THE UNIVERSITY of NEW MEXICO

Classes

- The C++ syntax for declaring a class is an extension of the structure concept found in C:

```
struct structure tag {
    private:
            private items member list
    public:
            public items member list
};
```

- Variables can be declared in the member lists, and unlike C, in C++ functions can also be declared in the member lists.
- All such variables are member variables, unless they are preceded by the keywords static, in which case they become a class variable.

Classes

- The scope of items in the *member list* can be modified using the **private** and **public** access specifiers:
  - **private** members can only be accessed by functions declared inside the structure, and by friend functions (discussed later).
  - **public** members may be directly accessed by elements external to the structure.
- Generally, member variables are placed in the private section of a class, and functions are declared in the public section. This support encapsulation!
- C++ also adds a new keyword called **class** that can be used in place of the keyword **struct**. There's only a slight difference between these two:
  - When **struct** is used, if access specifiers are not provided, all members default to public scope.
  - When **class** is used, if access specifiers are not provided, all members default to private scope.

THE UNIVERSITY of NEW MEXICO

Classes and ADTs

Ex. The Complex Number ADT:

```
class Complex {
   private:
      float real, imag;
   public:
      float get_real();
      float get_imag();
      void set_real(float);
      void set_imag(float);
      Complex add(Complex);
      Complex subtract(Complex);
      Complex multiply(Complex);
      Complex divide(Complex);
      // remaining member functions not shown
};
```

- The get_ and set_real and get_ and set_imag functions are called accessors, also known as getters and setters.

Classes and ADTs

- The previous class declaration would generally appear in a header file, e.g., complex.h
- The member function implementations should then go in a complex.cpp file, which should also contain an **#include <complex.h>** preprocessor directive.
- An implementation of the **get_real()** member function:

```
void Complex::set_real(float re) {real = re;}
```

- The double colon "::" operator is the scope resolution operator. It's used by the compiler to determine what class a given member function belongs to.
- Objects of type Complex can now be declared; for example,

```
Complex num1, num2, num3;
```

- creates three objects of type Complex.
- The statement

```
num1.set_real(3.2);
```

- assigns value 3.2 to member variable real in num1.

The this Pointer

- As demonstrated above, class member functions may refer to other members of their class directly without using the structure membership operators.
- They may also refer to their class members using the this pointer, a pointer that contains the address of the class object through which a member function has been invoked.
- When a member function accesses another member of its own class, the this pointer is implied. I.e., the member function implementation given above would be interpreted by the compiler as:

```
void Complex::set_real(float re)  {this->real = re;}
```

Inline functions

- It seems wasteful to invoke a function in order implement accessors; however, such an approach is necessary since we want **real** and **imag** to be a private variables.
- C++ provides a means, called inline functions, for eliminating the function call overhead associated with simple member functions.
- If a function is declared inline, the compiler attempts to replace every call of that function with a copy of the entire function. This is analogous to the substitution performed by the **#define** preprocessor directive.
- An inline function is declared by placing the keyword **inline** in front of the function definition, you don't need to change the function declaration. The function definition must also be moved to the header file.
- This is only a recommendation to the compiler, that it may ignore.

Inline functions

- The set real() inline function definition should be placed in the header file below the Complex class declaration:

```
inline void Complex::set real(float re)  {real = re;}
```

- C++ also provides a shorthand notation for specifying inline functions, which involves specifying the function body immediately after the function declaration in the class:

```
class Complex {
    private:
        float real, imag;
public:
        float get real() {return real;}
        float get imag() {return imag;}
        void set real(float re) {real = re;}
        void set imag(float im) {imag = im;}
// remaining members functions not shown
};
```

References

- The reference operator, &, is used to create a reference to a variable.
- This can be used to pass variables to a function by reference (If the reference operator is not used, then a function parameter is passed by value as in the C programming language.)
- The syntax for declaring a reference:

```
type& variable name = initialization expression;
```

where the reference must be initialized by an object of the type, or by one that can be converted to that type.

```
#include <iostream.h>
main() {
   int i=10;
   int& ref=i;                          // ref refers to i
   cout  << "ref = “ <<  ref;           // prints ref = 10
   ref = 20;
   cout << "i = “ <<  i;   // prints i = 20
}
```

References

- References are often applied to arguments and return values of functions. This allows us to perform call-by-reference.
- Passing an argument by reference to a function means that modifications made to the argument within the function change the actual argument, and not just a local copy of it.

Ex.

```
  void swap(int& x, int& y) {
        int temp;
        temp = x; x = y; y = temp;
     }
main() {
     int a=10, b=20;
     cout <<  "Before: a = “ <<  a << ", b = “ <<  b <<  endl;
     swap(a,b);
     cout  << "After: a = “ <<  a  << ", b = “ <<  b << endl;
}
```

What does this print?

```
main(), Before: a = 10, b = 20
main(), After: a = 20, b = 10
```

References

- What would the previous function print if we removed the &'s from the formal arguments in swap()? In this case, we'd be performing call-by-value, so copies of the actual parameters would be passed into cd swap(), and therefore the values would not appear swapped in main().
- References are often used to pass large objects as arguments, without incurring a lot of copying overhead. However, in this case it is very easy to create side-effects. To avoid this, it's common to pass the argument as a constant (using the keyword const). This will prevent that argument from being modified in the called function, and therefore prevents a side-effect in the calling function.
- Ex.

```
Complex add(const Complex&  z);
```

Friend Functions

- If a function declaration in a class is preceded by the keyword friend, then that function is said to be a friend of the class.
- A friend function is not an actual member of the class; however, it is still permitted to access the nonpublic data members of the class.
- In other words, the friend mechanism can be used to give nonmembers of a class the ability to access the nonpublic members of the class.
- Ex.

```
friend Complex add(const Complex&, const Complex&);
```

Constructors and Destructors

- C++ provides mechanisms for the developer to specify what should happen when a new object needs to be created (constructors), or an existing object needs to be destroyed (destructor).
- A constructor is any member function whose name is the same as the class name. If a constructor is supplied in a class definition, it is invoked whenever an object of that class needs to be created. E.g., when an object is declared, passed as an argument, or dynamically allocated using the new operator (the new operator is generally used in C++, rather than malloc() function as in C).
- A class may contain more than one constructor; however, each constructor must differ from the others in the class in either the type or the number of parameters (i.e., the constructors signatures must differ).

Constructors and Destructors

- Two forms of constructors have been given special names. The default constructor is called without any arguments, and the copy constructor is called with a single argument that has the same type as the class, which must passed to the constructor by reference.
- The default constructor is called whenever an array of objects needs to be created, and the copy constructor is called to make a copy of a class object.
- These two constructors are considered so basic that the complier will automatically generate them if they are not supplied by the class developer.
- It is important to note, however, that such complier-generated constructors will only create the class data members *without* initializing them.

Constructors and Destructors

- A destructor is used to destroy (i.e., reclaim) data pointed to by an object's data members immediately before the object itself is destroyed.
- A destructor is a member function whose name is the same as the class name prefixed with a tilde.
- A class may contain only one destructor, and this destructor function must have an empty parameter list. The destructor is automatically called whenever an object goes out of scope.
- In addition, the delete operator will invoke the destructor associated with an object, prior to reclaiming the object itself (the delete operator is generally used in C++, rather than the free() function as in C).
- A destructor is typically not needed in a class unless the objects of the class have data members that point to dynamically allocated memory.

Constructors and Destructors

Ex.

```
class Complex {
    private:
        float real, imag;
    public:
    // constructor
    Complex(float re=0, float im=0) {real=re; imag=im;}
    // destructor not needed
    };
```

This constructor supports the following declarations:

```
    Complex num1;               // num1.Re=0.0   num1.Im=0.0
    Complex num2(3.2);          // num2.Re=3.2   num2.Im=0.0
    Complex num3(5.1, 2.7);     // num3.Re=5.1   num3.Im=2.7
    Complex num3(, 2.7);        // error!
```

THE UNIVERSITY of NEW MEXICO

Function Overloading

- If two or more functions having the same scope share the same name, then this name is said to be overloaded.
- In C++, functions may share the same name as long as each function has a unique signature. A signature is considered unique if it differs from other signatures in either the number, type, or ordering of its arguments.
- Note that functions differing only in the return type may not share the same name.
- In C++, operators themselves can be overloaded - this is just a special type of function overloading called operator overloading.
- Most of the existing C++ operators can be overloaded; however, a new operator, such as ** for exponentiation, cannot be introduced.
- Other rules in C++: the meaning of operators supplied with built-in data types cannot be changed; the predefined precedence of operators is preserved when they are overloaded; unary operators must be overloaded as unary operators, and binary operators must be overloaded as binary operators.

THE UNIVERSITY of NEW MEXICO

Operator Overloading

- An operator function may either be a member function or a friend function; in both cases, at least one of the function arguments must be an object of the class itself. (Note however that the =, (), [], and -> operators may only be overloaded as member functions.)
- An operator function is defined in the same manner as other functions, except that the function name (in this case, an operator symbol) must be preceded by the keyword operator.
- Consider the previous class developed to represent complex numbers. The + and += operators can be overloaded to perform addition of two complex numbers.
- The next two slides show how this can be done with members functions, then friend functions.

THE UNIVERSITY of NEW MEXICO

Operator Overloading

Ex.

Operating overloading using member functions:

```
class Complex {
      private:
            float real, imag;
      public:
            Complex(float re=0, float im=0) {real=re; imag=im;}
            // operator overloading using member functions
            Complex operator+(const Complex&);
            Complex operator+=(const Complex&);
            // remaining member functions not shown
      };

      inline Complex Complex::operator+(const Complex& z)
            {return Complex(real + z.real, imag + z.imag);}

      inline Complex Complex::operator+=(const Complex& z)
            {real += z.real; imag += z.imag; return *this;}
```

Operator Overloading

Ex.

Operating overloading using friend functions:

```
class Complex {
        private:
                float real, imag;
        public:
                Complex(float re=0, float im=0) {real=re; imag=im; }

        // operator overloading using a friend function
        friend Complex operator+(const Complex&, const Complex&);
        friend Complex operator+=(Complex&, const Complex&);
        // remaining member functions not shown
};
inline Complex operator+(const Complex& z1, const Complex& z2)
        {return Complex(z1.real + z2.real, z1.imag + z2.imag);}
inline Complex operator+=(Complex& z1, const Complex& z2)
        {return Complex(z1.real += z2.real, z1.imag += z2.imag);}
```

THE UNIVERSITY of NEW MEXICO

Operator Overloading

Which one should you use? Consider the statement:

```
C = A+B;
```

where *A*, B, and *C* are Complex objects. The compiler expands the member function version as:

```
C = A.operator+(B);
```

and the friend function version as:

```
C = operator+(A,B);
```

Now, consider what happens if *A* is of type float. The member function form would yield an error, but the friend function would compile without error. For this reason, overloading operators using friend functions might be preferable.

Class Templates

- The ability to create parameterized classes is provided by the template facility in C++.
- A parameterized class is one in which the types of various members are specified using formal parameters when the class is declared, and actual values for these parameters are used to specify the missing type information when the class is used.
- The syntax requires the template keyword precede every forward declaration and definition of a template class. This keyword must be followed by a formal template parameter list that is surrounded by angle brackets.
- Ex.

```
template<class T>
class Matrix {
    private:
        // private members
    public:
        Matrix(int rdim, int cdim, const T& initval)
        // remaining member functions
};
```

Class Templates

- The use of the keyword class in the template parameter list indicates that the parameter that follows represents a type parameter.
- Thus, in the Matrix class, T represents a formal type parameter that can be used throughout the class declaration.
- Outside the class declaration, you must once again explicitly state the template parameters.
- For example, the constructor in declared in the Matrix class above would be defined using:

```
template<class T>
Matrix<T>::Matrix(int rdim, int cdim, const T& initval)
{
body of constructor
}
```

Inheritance

- The class syntax for C++ presented earlier was not quite complete. In order to incorporate the capability of inheriting existing classes, the syntax of the class specification must be modified. The complete syntax is as follows:

```
class class name : derivation list {
    private:
        private items member list
    protected:
        protected items member list
    public:
        public items member list
};
```

- where the derivation list specifies the base classes that the class being defined will inherit, as well as the access control that will be applied to these inherited classes.

Inheritance

- The protected members behave as public class members to derived classes; but to clients of the class, the protected members are private.
- The syntax for the derivation list is a comma separated list of the base class names prefixed by a keyword that controls the access to each base class:

  keyword base class name1, keyword base class name2, . . .

- The keywords that may be used in this derivation list are public and private. If a keyword is not given for a base class, it is assumed to be private.
- The inherited members of a public base class (i.e., a base class inherited using the public keyword) maintain their specified access levels within the derived class. E.g., private members of the base class may not be accessed by members of the derived class; however, protected and public members of the base class are also treated as protected and public members in the derived class

THE UNIVERSITY of NEW MEXICO

Inheritance

- In addition, any class that is derived from this derived class using the keyword public will also have access to the protected and public members of the original base class.
- Furthermore, objects created using the derived class will only have access to the public members of the base class.
- If a derived class is created using a private base class, then the protected and public members of the base class essentially become private members in the derived class.
- The derived class may access these members, but any other class derived from this derived class may not. Furthermore, objects created using the derived class will not have access to any members of the base class.

Inheritance

Ex.

```
class base {
      private:
            int x;
      protected:
            int y;
      public:
            int z;
};
class derived 1 : private base {
      private:
            int a;
      protected:
            int b;
      public:
            void assign x() {x=a;} // cannot access base::x
            void assign y() {y=a;} // o.k.
            void assign z() {z=a;} // o.k.
};
```

Inheritance

- Ex. (continued)

```
class derived 2 : public derived 1 {
        public:
                void print a() {cout  a;} // cannot access derived 1::a
                void print b() {cout  b;} // o.k.
                void print x() {cout  x;} // cannot access base::x
                void print y() {cout  y;} // cannot access base::y
                void print z() {cout  z;} // cannot access base::z
};
```

Polymorphism and Virtual Functions

- Recall that OO languages support polymorphism through the use of dynamic binding.
- In C++, polymorphic functions are called a virtual functions, and are declared in a base class using the keyword virtual in front of its declaration.
- As with other member functions, a virtual member function may be overloaded in derived classes by simply providing a new member function with the same name as the virtual member function in the base class. However, a member function that overloads a virtual member function is treated differently:
  - Use of the keyword virtual signifies that implementation of the member function provided in the base class is a default which will only be used by derived classes when they do not supply their own implementation.
  - If a derived class overloads the virtual function, then it will be used.

Polymorphism and Virtual Functions

- There's one more “trick” you need to know in order to make virtual functions work in C++. They must be invoked through pointers-to-objects.
- Specifically, if a pointer to an object of the base class is declared, then addresses of objects of the derived class may be assigned to this base class pointer. Now, if a virtual member function is invoked, the type (i.e., class) of the object that is currently pointed to by the base class pointer will determine which member function implementation is used.

Polymorphism and Virtual Functions

- Ex.

```
class base {
    private:
        int x;
    public:
        base() {x=5;}
        virtual void print() {cout <<  "x=“ <<  x << endl;}
};
class derived : base {
    private:
        int y;
    public:
        derived() {y=10;}
    // overload base::print()
    void print() {cout <<  "y=“ <<  y << endl;}
};
```

THE UNIVERSITY of NEW MEXICO

Polymorphism and Virtual Functions

- Ex. (continued)

```
main()
{
        base* bptr;
        base obj1;
        derived obj2;
        obj1.print();          // prints x=5
        obj2.print();          // prints y=10
        bptr = &obj1;
        bptr ->print();        // prints x=5
        bptr = &obj2;
        bptr->print();         // prints y=10
}
```

- Only the last two member function calls, invoked using the indirect membership operator, involve dynamic binding.