CS 213 Data Structures (1/2566)

C++ Review

Wirat Jareevongpiboon

- Yusuf Sahillioğlu, Department of Computer Engineering, Middle East Technical University
- Douglas Wilhelm Harder, M.Math.LEL, Department of Electrical and Computer Enginnering, University of Waterloo

C++

- Developed by Bjarne Stroustrup at Bell Labs
  - Called "C with classes"
  - C++ (increment operator) - enhanced version of C
- Improves on many of C's features
- Has object-oriented capabilities
- Superset of C
  - Can use a C++ compiler to compile C programs

Object Oriented Programming

1. Data Abstraction

– Providing only essential information to the outside world and hiding their background details.

Example

sort: you can sort an array with the C++ call, BUT you do not know the algorithm.

– In C++, we use **classes** to define our own abstract data types (ADT).

Object Oriented Programming

2. Information hiding

- Restrict access to data so that it can be manipulated only in authorized ways. Separate class declarations from implementation (e.g., public, private in C++).

3. Encapsulation

- bundling of data with the methods operating on that data.

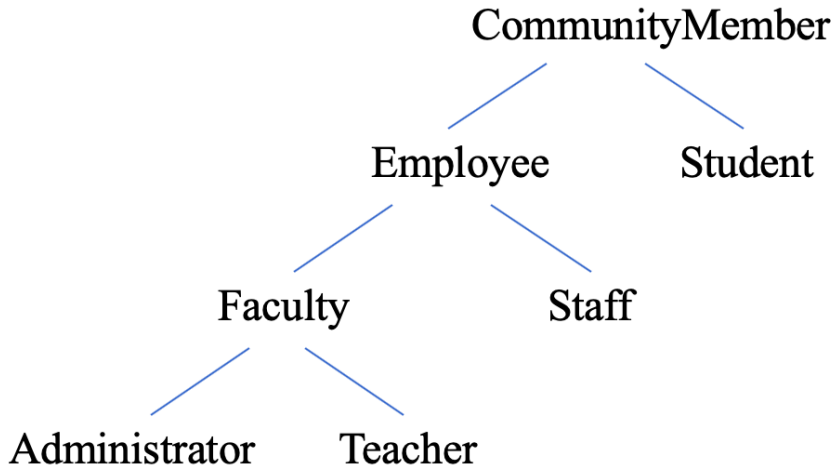
Object Oriented Programming

4. Inheritance

– Derive a new class (subclass) from an existing class (base class or superclass).

– Inheritance creates a hierarchy of related classes (types) which share code and interface.

Base Class	Derived Classes
Employee	Manager Researcher Worker
Account	CheckingAccount SavingAccount



A Basic C++ Program (1)

```
#include <iostream> //input/output
#include <math.h>   //header file for math

using namespace std;

int main()
{
    float x;

    cout << "Enter a real number: " << endl;
    cin >> x;  //scanf("%f", &x); in C

    cout << "The square root of " << x << " is: "
                << sqrt(x) << endl; //see comments part
}
```

A Basic C++ Program

```
cout << "Enter a real number: " << endl;
```

- In C++, all I/O is done by *classes*.
- A class is set up to handle input and output *streams*.
- Output to the screen is handled by the stream with standard name cout.
- This is a variable of class ostream. Similarly for cin.

A Basic C++ Program (2)

```
#include <iostream>

using namespace std;

int main()
{
    int a=23;
     int b=34;

    cout << "Enter two integers:" << endl;
    cin >> a >> b;
    cout << endl;

    cout << "a + b =" << a+b << endl;
    return 0;
}
```

A Basic C++ Program (3)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   double a=15.2;
    double b=34.3434343;

   cout << fixed << showpoint;
   cout << setprecision(2); //2 digits after the dot
    cout << setw(6) << a << endl;
   cout << setw(7) << b << endl;

   return 0;
}
```

Namespaces

- **Variables defined:**
  - In functions are *local variables*
  - In classes are *member variables*
  - Elsewhere are *global variables*

- **Functions defined:**
  - In classes are *member functions*
  - Elsewhere are *global functions*

- In all these cases, the keyword **static** can modify the scope

Namespaces

- Global variables/variables cause problems, especially in large projects
  - Hundreds of employees
  - Dozens of projects
  - Everyone wanting a function init()

- In C++ , this is solved using namespaces

Namespaces

- A namespace adds an extra disambiguation between similar names

```
namespace nsp {
    int n = 4;
    double mean = 2.34567;

    void init() {
        // Does something...
    }
}
```

- There are two means of accessing these global variables and functions outside of this namespace:
  - The namespace as a prefix => `nsp::init()`
  - The using statement => `using namespace nsp;`

Namespaces

- You will only need this for the standard name space
  - All variables and functions in the standard library are in the **std** namespace

```
#include <iostream>
std::cout << "Hello world!" << std::endl;
```

```
#include <iostream>
using namespace std;
cout << "Hello world!" << endl;
```

Classes and Objects

- **Class:** a type definition that includes both
  - data properties, and
  - operations permitted on that data
- **Object:** a variable that
  - is declared to be of some Class
  - Therefore, “An object is an instance of a class.”

Class Syntax

- A class in C++ consists of its **members**.
  - A member can be either data or functions.
- The functions are called **member functions** (or **methods**)
- Each instance of a class is an **object**.
  - Each object contains the data components specified in class.
  - Methods/functions are used to act on an object.

Class syntax - Example

```
// A class for simulating an integer memory cell

class  IntCell
{
   public:
        IntCell( )
        { storedValue = 0; }

        IntCell(int initialValue )
        { storedValue = initialValue;}

        int read( )
        { return storedValue; }

        void write( int x )
        { storedValue = x;}

   private:
        int storedValue;
};
```

constructors

Class Members

- **Public** member is visible to all routines and may be accessed by any method in any class.
- **Private** member is not visible to non-class routines and may be accessed only by methods in its class.
- Typically,
  - Data members are declared private
  - Methods are made public
- Restricting access is known as ***information hiding***.

Constructors

- A **constructor** is a method that executes when an object of a class is declared and sets the initial state of the new object.
- A constructor
  - has the same name with the class,
  - no return type
  - has zero or more parameters (the constructor without an argument is the *default constructor*)
- A class may have more than one constructor

Extra Constructor Syntax

```
// A class for simulating an integer memory cell

class  IntCell
{
   public:
        IntCell( int initialValue = 0 )
            : storedValue( initialValue) { }

        int read( ) const
            { return storedValue; }

        void write( int x )
            { storedValue = x; }
   private:
        int storedValue;
};
```

Single constructor (instead of two)

Object Declaration

- In C++, an object is declared just like a primitive type.

```
#include <iostream>
using namespace std;
#include "IntCell.h"

int main()
{
   //correct object declarations
   IntCell m1;
   IntCell m2 ( 12 );
   IntCell *m3;

   // incorrect object declaration
   Intcell m4();     // this is a function declaration,
                     // not an object
```

Object use in driver program

```
    // program continues

    m1.write(44);
    m2.write(m2.read() +1);
    cout << m1.read() << "    " <<  m2.read() << endl;
    m3 = new IntCell;
    cout << "m3 = " << m3->read() << endl;
    return 0;
}
```

Example: Class Time

```
class Time {
public:
   Time( int = 0, int = 0, int = 0 );  //default
                                       //constructor
   void setTime( int, int, int ); //set hr, min, sec
   void printMilitary();      // print am/pm format
   void printStandard();      // print standard format

private:
   int hour;
   int minute;
   int second;
};
```

Declaring Time Objects

```
// Note that implementation of class Time not given
// here.

int main(){
   Time t1,     // all arguments defaulted
        t2(2), // min. and sec. defaulted
        t3(21, 34),   // second defaulted
        t4(12, 25, 42); // all values specified
    . . .
}
```

Class Interface and Implementation

- In C++, separating the class interface from its implementation is common.
  - The interface remains the same for a long time.
  - The implementations can be modified independently.
- The **interface** lists the class and its members (data and function prototypes) and describes what can be done to an object.
- The **implementation** is the C++ code for the member functions.

Separation of Interface and Implementation

- It is a good programming practice for large-scale projects to put the interface and implementation of classes in different files.
  - For small amount of coding, it may not matter.
- *Header File*: contains the interface of a class.
  - Usually ends with `.h` (an include file)
- *Source-code file*: contains the implementation of a class.
  - Usually ends with `.cpp`
- `.cpp` file includes the `.h` file with the preprocessor command `#include`.

```
#include "myclass.h"
```

Separation of Interface and Implementation

- A big complicated project will have files that contain other files.
  - There is a danger that an include file (.h file) might be read more than once during the compilation process.
    - It should be read only once to let the compiler learn the definition of the classes.
- To prevent a `.h` file to be read multiple times, we use preprocessor commands `#ifndef` and `#define`

Class Interface

IntCell.h

```
#ifndef _IntCell_H_
#define _IntCell_H_

class  IntCell
{
   public:
      IntCell( int initialValue = 0 );
      int read( ) const;
      void write( int x );
   private:
      int storedValue;
};
#endif
```

Class Implementation

IntCell.cpp

```
#include <iostream>
#include "IntCell.h"
using std::cout;

//Construct the IntCell with initialValue
IntCell::IntCell( int initialValue)
   : storedValue( initialValue) {}

//Return the stored value.
int IntCell::read( ) const
{
    return storedValue;
}

//Store x.
void IntCell::write( int x )
{
    storedValue = x;
}
```

Scope operator:
ClassName :: member

A driver program

```
#include <iostream>
#include "IntCell.h"
using std::cout;
using std::endl;

int main()
{
      IntCell m;  // or IntCell m(0);

      m.write (5);
      cout << "Cell content : " << m.read() << endl;

      return 0;
}
```

A program that uses IntCell in file *TestIntCell.cpp*

Destructors

- Member function of class
- Performs termination housekeeping before the system reclaims the object's memory
- Complement of the constructor
- Name is tilde (~) followed by the class name

  E.g. `~IntCell( );`
  `~ Time( );`

- Receives no parameters, returns no value
- One destructor per class

Destructors

- A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

Destructor Example

```
class IntCell{
   public:
       IntCell(int initialValue=0)
       { storedValue = new int (initialValue);}

       ~IntCell()
       { delete storedValue; }

       int read( ) const
       { return *storedValue; }

       void write( int x )    { *storedValue = x; }
   private:
       int *storedValue;
}
```

When are Constructors and Destructors Called

- Global scope objects
  - Constructors called before any other function (including main)
  - Destructors called when main terminates (or exit function called)
- Automatic local objects
  - Constructors called when objects defined
  - Destructors called when objects leave scope (when the block in which they are defined is exited)
- `static` local objects
  - Constructors called when execution reaches the point where the objects are defined
  - Destructors called when main terminates or the exit function is called

Accessor and Modifier Functions

- A method that examines but does not change the state of its object is an **accessor**.
  - Accessor function headings end with the word `const`
- A member function that changes the state of an object is a **mutator**.

Example: Complex Class

Complex.h

```
#ifndef _Complex_H
#define _Complex_H

using namespace std;
class Complex
{
  float re, im; // by default private
  public:
      Complex(float x = 0, float y = 0): re(x), im(y) { }

      Complex operator*(Complex rhs) const;
      float modulus() const;
      void print() const;
};

#endif
```

Expected Input and Output

1. When the real and imaginary part both are positive.

First Complex Number = 5 + 6i

Second Complex Number = 1 + 3i

Output= 6 + 9i

2. When any one of the imaginary or real part is negative.

First Complex Number = -4 + 6i

Second Complex Number = 5 + (-3i)

Output= 1 + 3i

3. When both real and imaginary parts are negative.

First Complex Number = -5 + -(6i)

Second Complex Number = -3 + -(5i)

Implementation of Complex Class

Complex.cpp

```
#include <iostream>
#include <cmath>
#include "Complex.h"

Complex Complex::operator*(Complex rhs) const
{
   Complex prod;
   prod.re = (re*rhs.re - im*rhs.im);
   prod.im = (re*rhs.im + im*rhs.re);
   return prod;
}
float Complex::modulus() const
{
    return sqrt(re*re + im*im);
}
void Complex::print() const
{
    std::cout << "(" << re <<"," << im << ")" << std::endl;
}
```

Using the class in a Driver File

TestComplex.cpp

```
#include <iostream>
#include "Complex.h"
int main()
{
    Complex c1, c2(1), c3(1,2);
    float x;
    // overloaded  * operator!!
    c1 = c2 * c3 * c2;

    x = sqrt(c1.re*c1.re + c1.im*c1.im);

    // To correct it, we use an authorized public function
    x = c1.modulus();

    c1.print();
     return 0;
}
```

The compiler will stop here, since the Re and Imag parts are private.

Function Overloading

- Function overloading:
  - Functions with same name and different parameters
  - Overloaded functions performs similar tasks
    - Function to square `int`s and function to square `float`s

```
int square( int x) {return x * x;}
float square(float x) { return x * x; }
```

  - Program chooses function by function name and parameter types

```
// Using overloaded functions
#include <iostream>
using std::cout;
using std::endl;
int square( int x ) { return x * x; }
double square( double y ) { return y * y; }

int main()
{
   cout << "The square of integer 7 is " << square( 7 )
        << "\nThe square of double 7.5 is " << square( 7.5 )
        << endl;

   return 0;
}
```