

Lecture 1

2nd Semester

Operator Overloading-Part 1

University of Anbar

College of Computer Science and Information Technology

Department of Computer Science

Object Oriented Programming

Second Class

Dr. Ruqayah R. Al-Dahhan



00:00.00



What is Operator Overloading?

- The mechanism of giving special meanings to an operator is known as operator overloading.
- In simple words, it means assigning additional job to an operator; relative to a specific class (**user-defined type**)
- Allows us to define the behavior of operators when applied to objects of a class
- Operator Overloading does not allow us to alter the meaning of operators when applied to built-in types(**i.e** none of their original meaning will be lost).
 - one of the operands must be an object of a class

Operator Overloading

- Operators are overloaded by creating **Operator Functions.**
- Overloading an operator :
 - Write function definition as normal
 - Function name is keyword **operator** followed by the symbol for the operator being overloaded
 - **operator+** used to overload the addition operator (+)

Operator Functions

- Operator functions can be either :
 - Member function of a class or
 - Non-member function (friend function).
- The way of writing operator functions differs between member and non-member functions.

General Format of Member Function

Prototype

returnType ***operator**** (***parameters***) ;



any type



keyword



operator symbol

OR

returnType ***classname::operator**** (***params***) ;



any type



keyword



operator symbol

- *Return type* may be whatever the operator returns

Operator Functions as Class Members vs. as friend Functions

- Operator functions as member functions
 - ✓ member function has no argument for unary operators and only one argument for binary operators
- Operator functions as non-member functions
 - Must be **friends** if needs to access private members
 - Friend function will have only one argument for unary operators and two for binary operators

Unary Operators

- Can overload as
 - member function with no arguments.
 - As a global function with one argument.
 - Argument must be class object or reference to class object.

Simple Example of Unary Operator Overloading


```

#include<iostream>
using namespace std;
class space
{int x;   int y;   int z;
public:
space(int r , int m, int c)
{x=r;    y=m;    z=c;  }
void operator-() ;//unary operator
void display() ;    };//end class
void space::display()
{cout<<x<<"    "<<y<<"    "<<z<<"    "<< endl;  }
void space:: operator-()
{x=-x;    y=-y;    z=-z;    }
main()
{space S(11,-22,33) ;
cout<<"s:   ";      S.display() ;
-S;                  cout<<"s:   ";      S.display() ;      }

```

Overloading Binary Operators

- member function with one argument.
- or
- Global function with two arguments:
 - One argument must be class object or reference to a class object.

Simple Example of Binary Operator Overloading

```

#include<iostream>
using namespace std;
class complex {
    double x, y;
public:
    complex() {      }
    complex(double r, double i)

        {x=r; y=i;}
    complex operator+(complex b);
    void display();
};
complex complex:: operator+(complex c)
    {complex temp;
    temp.x=x+c.x;
    temp.y=y+c.y;
    return(temp);
    }
void complex::display()
    {cout<<x<<"  +  "<<y<<"\n";
    }
main()
{complex c1,c2(1.6,2.7),c3;
c1=complex(2.5,3.5);
c3=c1+c2;
cout<<"c1  "<<"\n";c1.display();
cout<<"c2  "<<"\n";c2.display();
cout<<"c3  "<<"\n";c3.display();
}

```

2nd Semester

Lecture 2

Operator Overloading- Part2

University of Anbar

College of Computer Science and Information Technology

Department of Computer Science

Object Oriented Programming

Second Class

Dr. Ruqayah R. Al-Dahhan

Example1: An example of(++ operator and - - operator) functions (member functions) that operate on the object of **Check** class (object **m** in this case).

Sol:

```
#include <iostream>
using namespace std;
class Check
{ private:
    int count;
public:
    Check(){count=5;}
    void operator ++()
    {   count = count+1;   }
    void Display()
    { cout<<"Count: "<<count<< endl; }
    void operator --()
    {   count= count-1;   }
};
```

```
main()
{   Check m;
    ++ m ;    // calls "operator
++()" function
    m.Display();
    -- m ;
    m.Display();
}
```

Output:

Count: 6
Count: 5

Note:

++ operator operates on object **m** to increase the value of **data member count** by **1**.

- - operator operates on object **m** to

Overloading Binary Operators Using Friend Functions

- *Must precede with friend keyword, and declare a function class scope.*

Friend returnType operator* (parameters) ;



any type



keyword operator symbol

- *Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator.*
- *All the working and implementation would same as binary operator function(Member Function) except this function will be implemented outside of the class scope.*

Example2: An example of (+,-,*,/,==,>,<,unary -,>>,<< operator) functions (**Friend** functions) that operate on objects of **Money** class .

```
#include <cmath>
#include <iostream>
using namespace std;
class Money          // Class for amounts of money in US currency
{
    friend const Money operator+(const Money& amount1, const Money& amount2);
    friend const Money operator-(const Money& amount1, const Money& amount2);
    friend const Money operator*(const Money& amount1, const Money& amount2);
    friend const Money operator/(const Money& amount1, const Money& amount2);
    friend bool operator==(const Money& amount1, const Money& amount2);
    friend bool operator>(const Money& amount1, const Money& amount2);
    friend bool operator<(const Money& amount1, const Money& amount2);
```

```
    friend const Money operator-(const Money& amount);
    friend istream& operator>>(istream& istr, Money& amount);
    friend ostream& operator<<(ostream& ostr, const Money& amount);
public:
    Money();
    Money(int theDollars);
    Money(double amount);
    Money(int theDollars, int theCents);
private:
    int dollars;
    int cents;
    int dollarsPart(double amount) const; //private member function
    int  centsPart(double amount) const; //private member function
    int   round(double number) const; //private member function

};
```


//Addition operator

```
const Money operator+(const Money& amount1, const Money& amount2)
{
    int finalDollars= amount1.dollars+ amount2.dollars;
    int finalCents= amount1.cents+ amount2.cents;
    return Money(finalDollars, finalCents);
}
```

//Subtraction operator

```
const Money operator-(const Money& amount1, const Money& amount2)
{
    int finalDollars= amount1.dollars- amount2.dollars;
    int finalCents= amount1.cents- amount2.cents;
    return Money(finalDollars, finalCents);
}
```

//Multiplication operator

```
const Money operator*(const Money& amount1, const Money& amount2)
{
    int finalDollars= amount1.dollars * amount2.dollars;
    int finalCents= amount1.cents * amount2.cents;
    return Money(finalDollars, finalCents);
}
```

//Division operator

```
const Money operator/(const Money& amount1, const Money& amount2)
{
    int finalDollars= amount1.dollars / amount2.dollars;
    int finalCents= amount1.cents / amount2.cents;
    return Money(finalDollars, finalCents);
}
```

//Equal to operator

```
bool operator==(const Money& amount1, const Money& amount2)
{
    return ( (amount1.dollars == amount2.dollars) && (amount1.cents ==
amount2.cents) );
}
```

//Greater than operator

```
bool operator>(const Money& amount1, const Money& amount2)
{
    return ( (amount1.dollars > amount2.dollars)&& (amount1.cents >
amount2.cents) );
}
```

//Less than operator

```
bool operator<(const Money& amount1, const Money& amount2)
{ return ( (amount1.dollars < amount2.dollars)&& (amount1.cents < amount2.cents)
);}
```

//Unary subtraction operator

```
const Money operator-(const Money& amount)
{ return Money(-amount.dollars, -amount.cents);
}

istream& operator>>(istream& istr, Money& amount)
{ char dollarSign;
  istr >> dollarSign;
  if (dollarSign != '$') // if (strcmp(dollarSign, '$') == 0)
    { cout << "No dollar sign in Money input. \n";
      exit(1); }
  double amountAsDouble;
  istr >> amountAsDouble;
  amount.dollars = amount.dollarsPart(amountAsDouble);
  amount.cents = amount.centsPart(amountAsDouble);
  return istr; }
```

```
ostream& operator<<(ostream& ostr, const Money& amount)
```

```
{ int absDollars = abs(amount.dollars);
  int absCents = abs(amount.cents);
  ostr << "Account balance: ";
  if (amount.dollars < 0 || amount.cents < 0)
    ostr << "$";
  else
    ostr << '$' << amount.dollars;
  if (absCents >= 10)
    ostr << "." << absCents << endl;
  else
    ostr << "." << '0' << absCents << endl;
  return ostr;
}
```

// Constructors

```
Money :: Money()
{ dollars = 0.0;
  cents = 0.0;}
```

```

Money :: Money(double amount)
{ dollars = dollarsPart(amount);
  cents  = centsPart(amount);
}

Money :: Money(int theDollars)
{ dollars = theDollars;
  cents  = 0.0;
}

Money :: Money(int theDollars, int theCents)
{ if ( (theDollars < 0 && theCents > 0) || (theDollars > 0 && theCents < 0) )
  { cout << "Inconsistent money data.\n";
    exit(1);
  }
  dollars = theDollars;
  cents  = theCents;
}

// Private member functions
int Money :: dollarsPart(double amount) const
{ return static_cast<int> (amount);          //Use <cmath>          }

```

```

int Money :: centsPart(double amount) const

{ double doubleCents = amount * 100;
  int intCents      = (round(fabs(doubleCents)) );
  if (amount < 0) intCents = -intCents;
  return (intCents % 100); //Return the amount in 'cents'
}

int Money :: round(double number) const
{ return static_cast<int> (floor (number+0.5)); //Use <cmath>
}

int main()
{ Money baseAmount(100, 60), fullAmount;
  fullAmount = baseAmount + 25;
  cout << fullAmount << endl;
  //fullAmount = 25 + baseAmount;
  Money yourAmount, myAmount(10,9);
  cout << "Enter an amount of money, (use the dollar sign in front): ";

```

```
cin >> yourAmount;
cout << "Your amount is: " << yourAmount << endl;
fullAmount = yourAmount + 25.34;
cout << "Your amount + 25.34 is: " << fullAmount << endl;
fullAmount = yourAmount - 70.78;
cout << "Your amount - 70.78 is: " << fullAmount << endl;
fullAmount=baseAmount+fullAmount;
cout << "fullAmount: " << fullAmount << endl;
return 0;
}
```

Output

When the previous code (i.e Example2) is compiled and executed, it produces the following results:

Account balance: \$125.60

Enter an amount of money, (use the dollar sign in front): \$567.93

Your amount is: Account balance: \$567.93

Your amount + 25.34 is: Account balance: \$592.127

Your amount - 70.78 is: Account balance: \$521.347

Notes

- C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.
- Here, it is important to make operator overloading function a friend of the class because it would be called without creating an

Notes

- *Static Cast: This is the simplest type of cast which can be used. It is a **compile time cast**. It does things like implicit conversions between types (such as int to float), and it can also call explicit conversion functions (or implicit ones).*

static_cast <new_type> (expression) → Returns a value of type new_type.

- *The **floor()** function in C++ returns the largest possible integer value which is less than or equal to the given argument.*
- ***fabs()** function is a library function of cmath header, it is used to find the absolute **value** of the given number, it accepts a number and returns absolute **value**.*

ASSIGNMENT

Submission Deadline: 23rd March 2020

1- Re-write the Example 1(in this lecture) to overload the operators ++ and -- using friend functions instead of the member functions.

2- Write a class **Person** with a couple of private members ((String) **Name** and (integer) **Age**) to overload the stream insertion >> and extraction operators << using a friend function.

Lecture 3

2nd Semester

Inheritance -Part 1

University of Anbar

College of Computer Science and Information Technology

Department of Computer Science

Object Oriented Programming

Second Class

Dr. Ruqayah R. Al-Dahhan

Inheritance

- One of the most **important** concepts in **object-oriented programming** is that **inheritance**. Inheritance allows us to **define a class in terms of another class**, which makes it easier to create and maintain an application. This also provides an opportunity to **reuse the code** functionality and **fast implementation** time
- We frequently **classify** objects.

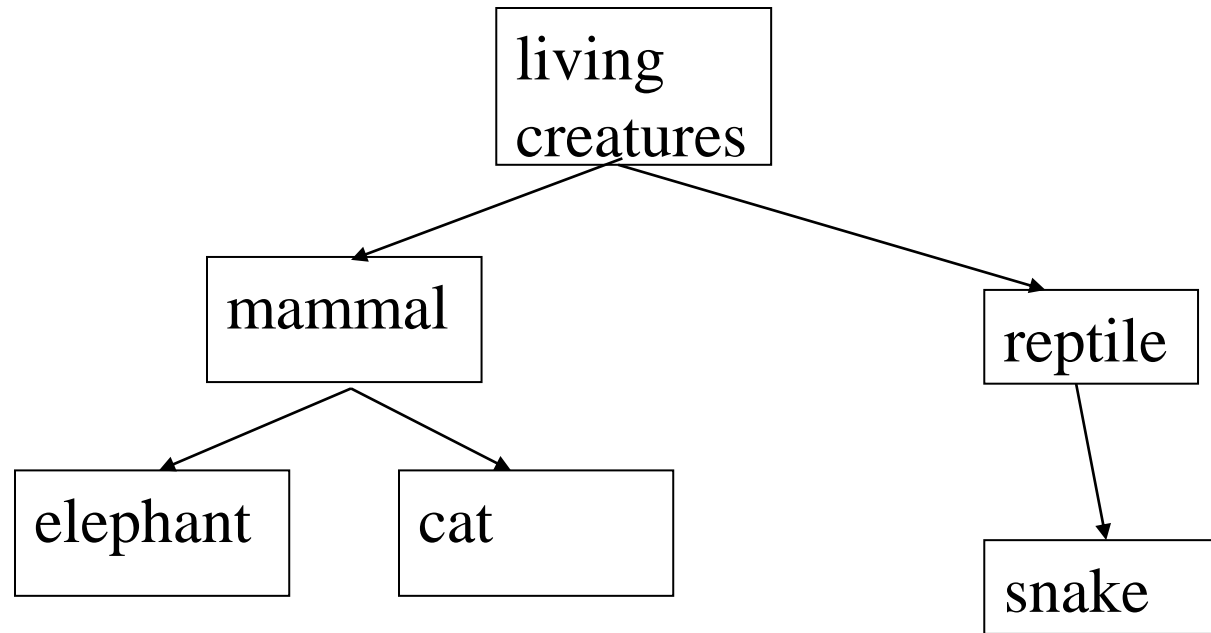
Mammals are

- warm-blooded**
- higher vertebrates.**

This information is valid for elephant and mouse but is best if we only have to express it only once for all mammals and not have to duplicate for every mammal.

Large body of knowledge can be presented in a compact way.

Inheritance



The class **cat** is **derived** from the class **mammal**. We say that a cat “**is a** mammal” and is also a “living creature”. But a cat “is not a reptile”. It is often useful to build our Object-Oriented programs this way.

Inheritance

- Inheritance is the capability of one class to acquire properties and characteristics from another class.
- The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class.
- The class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

How to define a derived class?

- To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

Class Derived_{class}: Access-Specifier Base_{class}

↑

↑

Keyword (***Public***, ***Private***, ***Protected***)

- If the access-specifier is not used, then it is private by default.

Example: Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>
using namespace std;
// Base class
class Shape {
public:
    void setWidth(int w) {
        width = w;
    }
    void setHeight(int h) {
        height = h;
    }
protected:
    int width;
    int height;
};
```

// Derived class

```
class Rectangle: public Shape {
public:
    int getArea() {
        return (width * height);
    }
};
main() {
    Rectangle Rect;
    Rect.setWidth(5);
    Rect.setHeight(7);

    // Print the area of the object.
    cout << "Total area: " << Rect.getArea() <<
endl;

    return 0;
}
```

Notes

- The **Output:** When the previous code is compiled and executed, it produces the following result:

Total area: 35

- The public inheritance relations between the classes:
 - Shape is the base class, and Rectangle is derived from it.
- We can group information (and avoid having to duplicate code) in a way that reflects our application and the real things or ideas that our code models.
- In main, a **Rect** object can call any functions in the Base class (**Shape**) that is derived from.

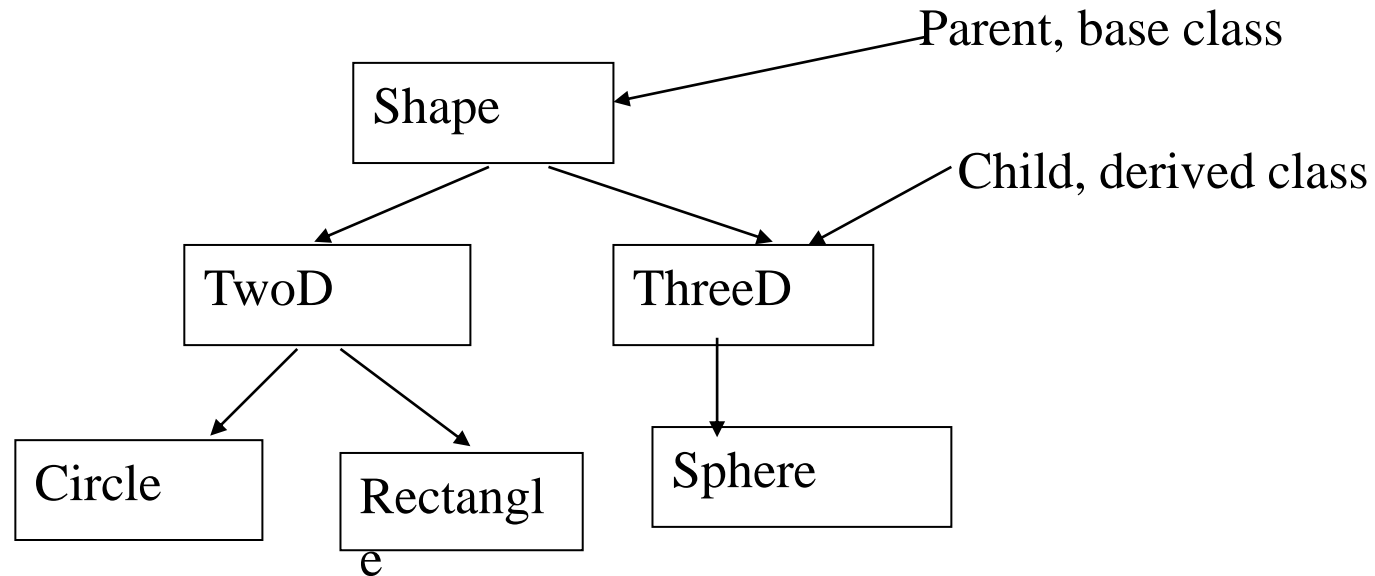
2nd Semester

Lecture 4

Inheritance -Part 2

University of Anbar
College of Computer Science and Information Technology
Department of Computer Science
Object Oriented Programming
Second Class
Dr. Ruqayah R. Al-Dahhan

Inheritance in C++



How can we code our objects like this?


```
#include <iostream>
Using namespace std;
const static double PI = 3.141592654;
```

```
class Shape{ // base class
private:
    char * label; // a string label for the shape
};
class TwoD : public Shape{ // two dimensional shapes
private:
    double x, y;
};
class ThreeD : public Shape{ // three dimensional shapes
private:
    double x, y, z;
};
class Circle : public TwoD{ // Circle extends TwoD
public:
    double area(){ return PI * radius * radius; }
private:
    double radius;
};
```

```
class Rectangle : public TwoD{
public:
    double area(){ return width * height; }
private:
    double width, height;
};

class Sphere : public ThreeD{
public:
    double volume(){ return 3.0 / 4.0 * PI * radius * radius
                    * radius; }
private:
    double radius;
};
int main(){
    Sphere s1;
    Circle c1;
    Shape *shptr;
    shptr = & s1;
    shptr = & c1;
    return 0;
}
```

Notes

- No output - this is only a skeleton code.
- The public inheritance relations between the classes.
- Shape is the base class, and TwoD and ThreeD are derived from it.
- We can group information (and avoid having to duplicate code) in a way that reflects our application and the real things or ideas that our code models.
- Note that in main, a Shape pointer can point to any object that is derived from it.

Inheritance

For the parent (of class Circle) we might write:

```
class TwoD : public Shape{
    public:
        void print() {cout<< x;}
        //..
    protected:
        double x,y;
        //..
};
```

protected: data of the class is accessible to the derived classes (and friends), but not any other part of the program.

Inheritance

For the parent class of Circle we write:

```
class TwoD:public Shape{
    public:
        void print() {cout<<...;}
        //..
    protected:
        double x,y;
        //..
};
```

For the derived class we write:

```
class Circle:public TwoD{
    public:
        double area();
    private:
        double radius;
};
```

public print() and protected x and y, are also part of Circle - inherited from the parent.

Inheritance

```
class Circle:public TwoD{  
    public:  
        double area();  
    private:  
        double radius;  
};
```

This is called **public inheritance**.

public and protected data of the parent class are inherited in the derived class and have the same access type. (We can have private and protected inheritance as well. Public inheritance is the most commonly used.)

```

#include <iostream>
Using namespace std;
const static double PI = 3.141592654;

class Shape{ // base class
private:
    char * label; // a string label for the shape
};

class TwoD : public Shape{ // two dimensional
    shapes
protected:
    double x, y;
};

class ThreeD : public Shape{ // three dimensional
    shapes
protected:
    double x, y, z;
};

class Circle : public TwoD{ // Circle extends TwoD
public:
    double area(){ return PI * radius * radius; }
private:
    double radius;
};

```

```

class Rectangle : public TwoD{
public:
    double area(){ return width * height; }
private:
    double width, height;
};

class Sphere : public ThreeD{
public:
    double volume(){ return 4.0 / 3.0 * PI * radius * radius *
        radius; }
private:
    double radius;
};

int main(){
    Sphere s1;
    Circle c1;
    cout << "size of s1 is " << sizeof(s1) << endl;
    cout << "size of c1 is " << sizeof(c1) << endl;
    return 0;
}

```

Notes

- **Output:**

size of s1 is 36

size of c1 is 28

$$36 = 4 (\text{char} *) + 3 * 8 (\text{double}) + 8 (\text{double})$$

$$28 = 4 (\text{char} *) + 2 * 8 (\text{double}) + 8 (\text{double})$$

Circles inherit label and x and y

Spheres inherit label, x,y and z

2nd Semester

Lecture 5

Inheritance -Part 3

University of Anbar
College of Computer Science and Information Technology
Department of Computer Science
Object Oriented Programming
Second Class
Dr. Ruqayah R. Al-Dahhan

Inheritance

```
class Student {  
    public:  
        Student(string s,int id);  
        void print();  
    protected:  
        string surname;  
        int studentID;  
};
```

We can create a new type
Graduate from Student with
added extra information. How do we
get initialisation of the information?

```
class GraduateStudent : public Student {  
    public:  
        GraduateStudent(string s,int id,string t);  
    protected:  
        string thesis;  
};
```

Inheritance

- Each class has its own constructor.
- The constructor for `Student` is placed in the initialiser list of the `GraduateStudent` constructor.
- ```
Student::Student(string s,int id) {surname=s; ID=id;}
```
- ```
GraduateStudent::GraduateStudent(string s, int id, string t)  
{Student(s,id);  
thesis=t; // code}
```
- **The constructor for `Student` (base class) is placed in the initialiser list of the `GraduateStudent` (derived class) constructor.**

Inheritance

- The derived class constructor does not always need to call the base class constructor.
- If the base class has a constructor with **defaulted** arguments, then the derived class does **not** need to call the constructor of the base class; otherwise it **must** call it.

```

#include <iostream>
#include <cstring>
using namespace std;
class Student{
public:Student(){ };
Student( string , int ); // normal initialiser
list
    GraduateStudent::GraduateStudent(string s, int id,
    string t)
    { Student (s,id);
    thesis=t; cout<< s << " " <<id <<"
    " <<thesis<<endl;}
    main()
    { Student s1( "Somename",001 );
    GraduateStudent gs1( "Distributed
    Computing",12, "Anothername" );

    protected:
    string surname;int ID;};
Student::Student(string s, int id=0)
{ surname=s; ID=id; }
class GraduateStudent : public Student{
public:
GraduateStudent( string , int , string );
private:
string thesis; // thesis topic
};

```

Access Control

- **Derived class can access all non-private members of the base class.** So if you do not want a **member function of** the base class to be accessed by the derived class should be declared as **private** in the base class.
- A derived class **inherits all the methods** of the base class, but **does not** include:
 - Base class constructor, destructor, and copy constructor.
 - Operator overloading the base class.
 - Friend base class function.

Summary

- **Inheritance** is the mechanism **of deriving new classes from old ones**.
- The keywords **public**, **private**, and **protected** are used as visibility modifiers for class members.

2nd Semester

Lecture 6

I/O Streams

University of Anbar
College of Computer Science and Information Technology
Department of Computer Science
Object Oriented Programming
Second Class
Dr. Ruqayah R. Al-Dahhan

- **cin** represent the input stream connected to the standard input device (usually the keyboard).
- **cout** represent the output stream connected to the standard output device (usually the screen).
- We use **cin** and **cout** objects for input and output data of various types.
- The reading for variable will be terminated at the encounter of a **white space** or a **character that doesn't match the destination type**.

Example

Ex:-

```
int code;
```

```
cin>>code;
```

if input

458D

- The operator will read the characters up to 8, the character **D** remains in the input stream and will be input to the next cin statement.

Put() and get() functions:-

- get() function defined in istream class.
- get() handle a single character.
- There two form : get(char*) and get(void).
- get() can fetch a charter including the blank space, tab, and the new line character.
- The get(char*) assign the input character to its argument.
- The get(void) return the input character.
- **Since these functions are members of I\O stream classes, we must invoke them using an appropriate object.**

Example

```
char c;  
  
cin.get(c) // or c = cin.get( );  
  
while ( c!= '\n')  
{  
  
    cout<<c;  
  
    cin.get( c );  
  
}
```

- **The above while loop will not work properly if `cin>> c;` is used instead `cin.get(c);` .**
- The function **put()**, a member of ostream class, can be used to output a line of text, character by character.

```

#include <iostream >
using namespace std;
main()
{
    int count = 0;
    char c;
    cout<<"Input Text \n";
    cin.get( c );
    while ( c != '\n')
    {    cout.put( c );
        count++;
        cin.get( c );    }
    cout<<"\n Number of characters= "<<count<<"\n"; }

```

- **When we type a line of input, the text is sent to the program as soon as we press Return key. (cin begin read from istream).**
- **The program then reads one character at a time using the statement **cin.get(c);** and display it using **cout.put(c).****

getline() and write() functions

- The **getline()** function reads a whole line of text that ends with a newline character (Return key). This function can be invoked using the object **cin** as follow:-

cin.getline (line, size);

- This invoke reads character input into the variable line, the reading is terminated as soon as either the newline is encountered or size -1 characters are read.
- The newline character is read but not saved, instead it is replaced by the null character.
 - Following example show use of **getline()** :

```
#include <iostream>

Using namespace std;

main ( )

{ int size =20 ;

char city [20]

cout<<"Enter city name \n";

cin>>city;

cout <<"city name : "<<city <<"\n\n";

cout<<"Enter city name agin : \n";

cin.getline (city , size );

cout <<"city name now : "<<city <<"\n\n";

cout <<"Enter another city name : \n";

cin.getline (city , size );

cout <<"New city name : "<<city <<"\n\n";

}
```

O/P:?

- **The write() function display an entire line and has the following form :-**
 - **cout . write (line , size);**
 - **line :** represents the name of the string to be displayed .
 - **size :** indicates the number of characters to display.
- *** if the size is greater than the length of line , then it display beyond the bound of line.**

```
#include <iostream >

#include<cstring >

Using namespace std

main( )

{ char *string1 ="C++";

char *string2 ="programming";

int m =strlen (string1);

int n = strlen (string2);

for (int i=1; i<n ; i++)

{ cout . write (string2 ,i); cout<<"\n"; }

for (i=n ; i>0 ; i--)

{ cout . write (string2 ,i); cout<<"\n"; }

cout . write (string1,m). write (string2,n);

cout<<"\n";

cout . write (string1,10 ); }
```