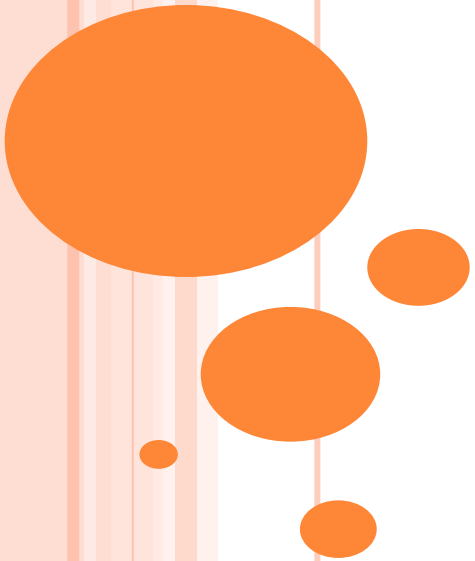


CSE 107: Object Oriented Programming Language

Dr. Tanzima Hashem
Professor
CSE, BUET

Function Overloading

- Two or more functions can share the same name as long as either
  - The type of their arguments differs, or
  - The number of their arguments differs, or
  - Both of the above
- The compiler will automatically select the correct version
- The return type alone is not a sufficient difference to allow function overloading

Function Overloading

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

class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};
```

Function Overloading

```
int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Function Overloading

```
int sum (int x, int y)
{
 cout << x+y;
}

double sum(double x, double y)
{
 cout << x+y;
}
```

Function Overloading

```
int sum (int x, int y)
{
 cout << x+y;
}

int sum(int x, int y, int z)
{
 cout << x+y+z;
}
```

Function Overloading

```
// This is incorrect and will not compile.
int f1 (int a);
double f1 (int a);
.
.
.
f1(10)          //which function  does the computer call???
```

Overloading Constructor Functions

- It is possible to overload constructors, but destructors cannot be overloaded

- Three main reasons to overload a constructor function
  - To gain flexibility
  - To support arrays
  - To create copy constructors

Overloading Constructor Functions

- Overloading constructor functions also allows the programmer to select the most convenient method to create objects
  - **Date d1(22, 9, 2007);** // uses Date(int d, int m, int y)
  - **Date d2("22-Sep-2007");** // uses Date(char* str)

- There must be a constructor function for each way that an object of a class will be created, otherwise compile-time error occurs
- Let, we want to write
  - **MyClass ob1, ob2(10);**
- Then MyClass should have the following two constructors (it may have more)
  - **MyClass ( ) { … }**
  - **MyClass (int n) { … }**
- Whenever we write a constructor in a class, the compiler does not supply the default no argument constructor automatically

Overloading Constructor Functions

- No argument constructor is also necessary for declaring arrays of objects without any initialization
  - **MyClass array1[5];**

    // uses MyClass () { … } for each element

- But with the help of an overloaded constructor, we can also initialize the elements of an array while declaring it
  - **MyClass array2[3] = {1, 2, 3}**

    // uses MyClass (int n) { … } for each element

- As dynamic arrays of objects cannot be initialized, the class must have a no argument constructor to avoid compiler error while creating dynamic arrays using “new”.

Copy Constructor

- Copy constructor is a constructor which is a constructor which creates an object by initializing it with an object of the same class

The copy constructor is used to:

- Initialize one object from another object of the same type in a declaration statement

```
MyClass y;
MyClass x = y;
```

- Copy an object to pass it as an argument to a function

```
func1(y); // calls "void func1( MyClass obj )"
```

- Copy an object to return it from a function

```
y = func2(); // gets the object returned from "MyClass func2()"
```

Copy Constructor

- If we do not write our own copy constructor, then the compiler supplies a copy constructor that simply performs bitwise copy
- If the class has pointer variables and dynamic memory allocations then bitwise copy is not enough
- We can write our own copy constructor to dictate precisely how members of one object should be copied to other
- The most common form of copy constructor is

```
classname (const classname &obj) {
    // body of constructor
}
```

Copy Constructor

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

class Rectangle {
        int *width, *height;
public:
        Rectangle(int, int);
        ~Rectangle ();
        int area () {return (*width * *height);}
};

Rectangle::Rectangle (int a, int b) {
        width= new int;
        height = new int;
        *width = a;
        *height = b;
}
```

Copy Constructor

```
Rectangle:: ~Rectangle () {
        delete width;
        delete height;
}

Rectangle larger(Rectangle recta, Rectangle rectb){
        if(recta.area()>rectb.area())
                return recta;
        else
                return rectb;
}
```

Copy Constructor

```
int main () {
        Rectangle recta (3,4);
        Rectangle rectb (5,6);

        Rectangle rectc=recta; //this will cause the program to crash

        Rectangle rect_larger(0,0);

        rect_larger=larger(recta, rectb); //this will cause the program to crash


        cout << "recta area: " << recta.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        cout << "rectc area: " << rectc.area() << endl;

        cout << "rect_larger area: " << rect_larger.area() << endl;


        return 0;
}
```

Copy Constructor

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

class Rectangle {
        int *width, *height;
public:
        Rectangle(int, int);
        Rectangle(const Rectangle &r);
        ~Rectangle ();
        int area () {return (*width * *height);}
};

Rectangle::Rectangle (int a, int b) {
        width= new int;
        height = new int;
        *width = a;
        *height = b;
}
```

Copy Constructor

```
Rectangle::Rectangle(const Rectangle &r){
        width=new int;
        height=new int;
        *width=*r.width;
        *height=*r.height;
}

Rectangle:: ~Rectangle () {
        delete width;
        delete height;
}

Rectangle larger(Rectangle recta, Rectangle rectb){
        if(recta.area()>rectb.area())
                return recta;
        else
                return rectb;
}
```

Copy Constructor

```
int main () {
        Rectangle recta (3,4);
        Rectangle rectb (5,6);

        Rectangle rectc=recta; //this will call copy constructor
        Rectangle rect_larger(0,0);

        rect_larger=larger(recta, rectb); // will cause error cause assignment operator -_-
         //this will call both copy constructor and destructor 3 times

        cout << "recta area: " << recta.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        cout << "rectc area: " << rectc.area() << endl;


        cout << "rect_larger area: " << rect_larger.area() << endl;


        return 0;
}
```

Using Default Arguments

- Related to function overloading
  - Essentially a shorthand form of function overloading

- Allows to give a parameter a default value when no corresponding argument is specified when the function is called
  - **void f1(int a = 0, int b = 0) { … }**
  - It can now be called in three different ways
    - **f1(); // inside f1() 'a' is '0' and b is '0'**
    - **f1(10); // inside f1() 'a' is '10' and b is '0'**
    - **f1(10, 99); // inside f1() 'a' is '10' and b is '99'**
  - We cannot give 'b' a new (non-default) value without specifying a new value for 'a'
  - While specifying non-default values, we have to start from the leftmost parameter and move to the right one by one

Using Default Arguments

- Default arguments must be specified only once: either in the function's prototype or in its definition

- All default parameters must be to the right of any parameters that don't have defaults
  - **void f2(int a, int b = 0); // no problem**
  - **void f3(int a, int b = 0, int c = 5); // no problem**
  - **void f4(int a = 1, int b); // compiler error**

- Default arguments must be constants or global variables.
- Default arguments cannot be local variables or other parameters

Using Default Arguments

- Relation between default arguments and function overloading
  - **void f1(int a = 0, int b = 0) { … }**
  - It acts as the same way as the following overloaded functions –
    - **void f2( ) { int a = 0, b = 0; … }**
    - **void f2(int a) { int b = 0; … }**
    - **void f2(int a, int b) { … }**
- Constructor functions can also have default arguments
- It is possible to create copy constructors that take additional arguments, as long as the additional arguments have default values
  - **MyClass(const MyClass &obj, int x = 0) { … }**

Overloading and Ambiguity

- Due to automatic type conversion rules
- Example 1:
  - **void f1(float f) { … }**
  - **void f1(double d) { … }**
  - **float x = 10.09;**
  - **double y = 10.09;**
  - **f1(x); // unambiguous – use f1(float)**
  - **f1(y); // unambiguous – use f1(double)**
  - **f1(10); // ambiguous, compiler error**
    - Because integer '10' can be promoted to both "float" and "double".

Overloading and Ambiguity

- Due to the use of reference parameters
- Example 2:
  - **void f2(int a, int b) { … }**
  - **void f2(int a, int &b) { … }**
  - **int x = 1, y = 2;**
  - **f2(x, y); // ambiguous, compiler error**

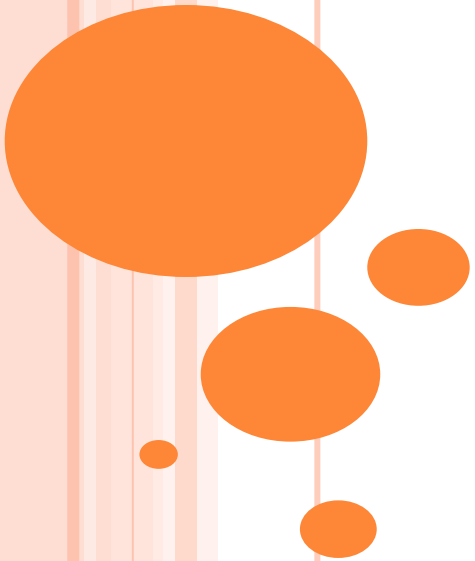
Overloading and Ambiguity

- Due to the use of default arguments
- Example 3:
  - **void f3(int a) { … }**
  - **void f3(int a, int b = 0) { … }**
  - **f3(10, 20); // unambiguous – calls f3(int, int)**
  - **f3(10); // ambiguous, compiler error**

Finding the address of an overloaded function

- Example:
  - **void space(int a) { … }**
  - **void space(int a, char c) { … }**
  - **void (*fp1)(int);**
  - **void (*fp2)(int, char);**
  - **fp1 = space; // gets address of space(int)**
  - **fp2 = space; // gets address of space(int, char)**

- So, it is the declaration of the pointer that determines which function's address is assigned



Acknowledgement

http://faizulbari.buet.ac.bd/Courses.html

http://mhkabir.buet.ac.bd/cse201/index.html

The End

Topic Covered: Chapter 5 (except 5.3)