Function Overloading

Chapter 5

Department of CSE, BUET

Objectives

- Overloading Constructor Functions
- Creating and Using a Copy Constructor
- The **overload** Anachronism (not in syllabus)
- Using Default Arguments
- Overloading and Ambiguity
- Finding the address of an overloaded function

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
- There must be a constructor function for each way that an object of a class will be created, otherwise compile-time error occurs.

Overloading Constructor Functions (contd.)

- 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.
- 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

Overloading Constructor Functions (contd.)

- 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)
- Another reason to overload a constructor function is to support dynamic arrays of objects created using "new".
- 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".

Creating and Using a Copy Constructor

- By default when a assign an object to another object or initialize a new object by an existing object, a bitwise copy is performed.
- This cause problems when the objects contain pointer to dynamically allocated memory and destructors are used to free that memory.
- It causes the same memory to be released multiple times that causes the program to crash.
- Copy constructors are used to solve this problem while we perform object initialization with another object of the same class.
  - MyClass ob1;
  - MyClass ob2 = ob1; // uses copy constructor
- Copy constructors do not affect assignment operations.
  - MyClass ob1, ob2;
  - ob2 = ob1; // does not use copy constructor

Creating and Using a Copy Constructor (contd.)

- If we do not write our own copy constructor, then the compiler supplies a copy constructor that simply performs bitwise copy.
- We can write our own copy constructor to dictate precisely how members of two objects should be copied.
- The most common form of copy constructor is
  - classname (const classname &obj) {
  - // body of constructor
  - }

Creating and Using a Copy Constructor (contd.)

- Object initialization can occur in three ways
  - When an object is used to initialize another in a declaration statement
    - MyClass y;
    - MyClass x = y;
  - When an object is passed as a parameter to a function
    - func1(y); // calls "void func1(MyClass obj)"
  - When a temporary object is created for use as a return value by a function
    - y = func2(); // gets the object returned from "MyClass func2()"
- See the examples from the book and the supplied codes to have a better understanding of the activities and usefulness of copy constructors.
  - Example: copy-cons.cpp

Using Default Arguments

- It is related to function overloading.
  - Essentially a shorthand form of function overloading
- It 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 can see that we cannot give ‘b’ a new (non-default) value without specifying a new value for ‘a’.
  - So while specifying non-default values, we have to start from the leftmost parameter and move to the right one by one.

Using Default Arguments (contd.)

- 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
- So, once you begin to define default parameters, you cannot specify any parameters that have no defaults.
- Default arguments must be constants or global variables. They cannot be local variables or other parameters.

Using Default Arguments (contd.)

- 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.

Using Default Arguments (contd.)

- 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) { … }
- This flexibility allows us to create copy constructors that have other uses.
- See the examples from the book to learn more about the uses of default arguments.

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 (contd.)

- 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 (contd.)

- 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.

Lecture Contents

- Teach Yourself C++
  - Chapter 5 (Full, with exercises)
    - Except "The **overload** Anachronism"