A Closer Look at Classes

ASSIGNING OBJECTS

- One object can be assigned to another provided that both objects are of the same type.
- It is not sufficient that the types just be physically similar – their type names must be the same.
- By default, when one object is assigned to another, a bitwise copy of all the data members is made. Including compound data structures like arrays.
- Creates problem when member variables point to dynamically allocated memory and destructors are used to free that memory.
- Solution: **Copy constructor** (to be discussed later)
- **Example:** assign-object.cpp

Passing Objects to Functions

- Objects can be passed to functions as arguments in just the same way that other types of data are passed.
- By default all objects are passed by value to a function.
- Address of an object can be sent to a function to implement call by reference.
- **Examples:** From book

Passing Objects to Functions

- In call by reference, as no new objects are formed, constructors and destructors are not called.
- But in call value, while making a copy, constructors are not called for the copy but destructors are called.
- Can this cause any problem in any case?
- Yes. Solution: **Copy constructor** (discussed later)
- **Example**: obj-passing1.cpp, obj-passing2.cpp, obj-passing-problem.cpp

RETURNING OBJECTS FROM FUNCTIONS

- The function must be declared as returning a class type.
- When an object is returned by a function, a temporary object (invisible to us) is automatically created which holds the return value.
- While making a copy, constructors are not called for the copy but destructors are called
- After the value has been returned, this object is destroyed.
- The destruction of this temporary object might cause unexpected side effects in some situations.
- Solution: **Copy constructor** (to be discussed later)
- **Example:** ret-obj-1.cpp, ret-obj-2.cpp, ret-obj-3.cpp

FRIEND FUNCTIONS

- A friend function is not a member of a class but still has access to its private elements.
- A friend function can be
  - A global function not related to any particular class
  - A member function of another class
- Inside the class declaration for which it will be a friend, its prototype is included, prefaced with the keyword friend.
- Why friend functions ?
  - Operator overloading
  - Certain types of I/O operations
  - Permitting one function to have access to the private members of two or more different classes

Friend Functions

```
class MyClass
{
   int a; // private member
public:
   MyClass(int a1) {
      a = a1;
   }
   friend void ff1(MyClass obj);
};
```

```
// friend keyword not used
void ff1(MyClass obj)
{
   cout << obj.a << endl;
     // can access private
    member ‘a’ directly
   MyClass obj2(100);
   cout << obj2.a << endl;
}
void main()
 {
   MyClass o1(10);
   ff1(o1);
}
```

FRIEND FUNCTIONS

- A friend function is not a member of the class for which it is a friend.
  - MyClass obj(10), obj2(20);
  - obj.ff1(obj2); // wrong, compiler error
- Friend functions need to access the members (private, public or protected) of a class through an object of that class. The object can be declared within or passed to the friend function.
- A member function can directly access class members.
- A function can be a member of one class and a friend of another.
- **Example** : friend1.cpp, friend2.cpp, friend3.cpp

FRIEND FUNCTIONS

```
class YourClass; // a forward
   declaration
class MyClass {
  int a; // private member
public:
  MyClass(int a1) { a = a1; }
  friend int compare
   (MyClass obj1, YourClass
   obj2);
};
class YourClass {
  int a; // private member
public:
  YourClass(int a1) { a = a1; }
```

```
friend int compare (MyClass
   obj1, YourClass obj2);
};
void main() {
  MyClass o1(10); YourClass
   o2(5);
  int n = compare(o1, o2); // n = 5
}

int compare (MyClass obj1,
   YourClass obj2) {
  return (obj1.a – obj2.a);
}
```

Friend Functions

```
class YourClass; // a forward
   declaration
class MyClass {
  int a; // private member
public:
  MyClass(int a1) { a = a1; }
  int compare (YourClass obj) {
    return (a – obj.a)
  }
};
```

```
class YourClass {
  int a; // private member
public:
  YourClass(int a1) { a = a1; }
  friend int MyClass::compare
   (YourClass obj);
};
void main() {
  MyClass o1(10); Yourclass
   o2(5);
  int n = o1.compare(o2); // n = 5
}
```

Conversion Function

- Used to convert an object of one type into an object of another type.
- A conversion function automatically converts an object into a value that is compatible with the type of the expression in which the object is used.
- General form: *operator type() {return value;}*
- *type* is the target type and *value* is the value of the object after conversion.
- No parameter can be specified.
- Must be a member of the class for which it performs the conversion.
- **Examples**: From Book.

CONVERSION FUNCTION

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

class coord
{
    int x, y;
public:
    coord(int i, int j){ x = i; y = j; }
    operator int() { return x*y; }
};
```

```
int main
{
    coord o1(2, 3), o2(4, 3);
    int i;

    i = o1;
    // automatically converts to integer
    cout << i << ‘\n’;

    i = 100 + o2;
    // automatically converts to integer
    cout << i << ‘\n’;

    return 0;
}
```

Conversion Function

- Suppose we have the following two classes:
  - Cartesian Coordinate: CCoord
  - Polar Coordinate: PCoord

- Can we use conversion function to perform conversion between them?

```
CCoord c(10, 20);
PCoord p(15, 120);

p = c;
c = p;
```

STATIC CLASS MEMBERS

- A class member can be declared as ***static***
- Only one copy of a ***static*** variable exists – no matter how many objects of the class are created
  - All objects share the same variable
- It can be private, protected or public
- A ***static*** member variable exists before any object of its class is created
- In essence, a ***static*** class member is a global variable that simply has its scope restricted to the class in which it is declared

STATIC CLASS MEMBERS

- When we declare a ***static*** data member within a class, we are not defining it
- Instead, we must provide a definition for it elsewhere, outside the class
- To do this, we re-declare the ***static*** variable, using the scope resolution operator to identify which class it belongs to
- All ***static*** member variables are initialized to **0** by default

STATIC CLASS MEMBERS

- The principal reason ***static*** member variables are supported by C++ is to avoid the need for global variables
- Member functions can also be ***static***
  - Can access only other ***static*** members of its class directly
  - Need to access ***non-static*** members through an object of the class
  - Does not have a ***this*** pointer
  - Cannot be declared as ***virtual***, ***const*** or ***volatile***
- ***static*** member functions can be accessed through an object of the class or can be accessed independent of any object, via the class name and the scope resolution operator
  - Usual access rules apply for all ***static*** members
- **Example**: static.cpp

Static class members

```
class myclass {
   static int x;
public:
   static int y;
   int getX() { return x; }
   void setX(int x) {
      myclass::x = x;
   }
};
int myclass::x = 1;
int myclass::y = 2;
```

```
void main ( ) {
   myclass ob1, ob2;
   cout << ob1.getX() << endl; // 1
   ob2.setX(5);
   cout << ob1.getX() << endl; // 5
   cout << ob1.y << endl; // 2
   myclass::y = 10;
   cout << ob2.y << endl; // 10
   // myclass::x = 100;
   // will produce compiler error
}
```

CONST MEMBER FUNCTIONS AND *MUTABLE*

- When a class member is declared as *const* it can't modify the object that invokes it.
- A *const* object can't invoke a non-*const* member function.
- But a *const* member function can be called by either *const* or non-*const* objects.
- If you want a *const* member function to modify one or more member of a class but you don't want the function to be able to modify any of its other members, you can do this using *mutable*.
- *mutable* members can modified by a *const* member function.
- **Examples**: From Book.

Lecture Contents

- **Teach Yourself C++**
  - Chapter 3 (Full, with exercises)
  - Chapter 13 (13.2,13.3 and 13.4)