Chapter 11: More About Classes and Object-Oriented Programming

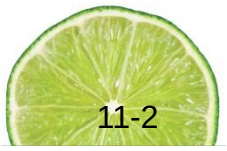
Starting Out with C++
Early Objects
Eighth Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

Addison-Wesley is an imprint of PEARSON

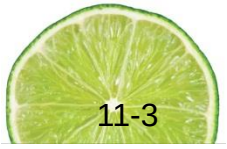
Topics

11.1 The `this` Pointer and Constant Member Functions
11.2 Static Members
11.3 Friends of Classes
11.4 Memberwise Assignment
11.5 Copy Constructors
11.6 Operator Overloading
11.7 Type Conversion Operators



Topics (continued)

11.8 Convert Constructors



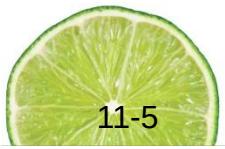
11.1 The `this` Pointer and Constant Member Functions

- **`this`** pointer:
  - Implicit parameter passed to a member function
  - points to the object calling the function
- **`const`** member function:
  - does not modify its calling object

Using the `this` Pointer

Can be used to access members that may be hidden by parameters with the same name:

```
class SomeClass
{
  private:
    int num;
  public:
    void setNum(int num)
    { this->num = num; }
};
```



Constant Member Functions

- Declared with keyword **`const`**

- When **`const`** appears in the parameter list,

```
int setNum (const int num)
```

the function is prevented from modifying the parameter. The parameter is read-only.

- When **`const`** follows the parameter list,

```
int getX()const
```

the function is prevented from modifying the object.

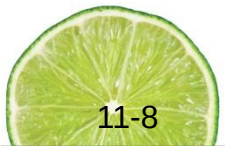
11.2 Static Members

- Static member variable:
  - One instance of variable for the entire class
  - Shared by all objects of the class
- Static member function:
  - Can be used to access static member variables
  - Can be called before any class objects are created

Static Member Variables

1) Must be declared in class with keyword **`static`**:

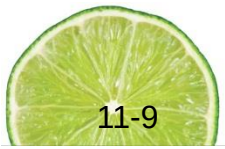
```
class IntVal
{
 public:
    intVal(int val = 0)
    { value = val; valCount++ }
    int getVal();
    void setVal(int);
 private:
    int value;
    static int valCount;
};
```



Static Member Variables

2) Must be defined outside of the class:

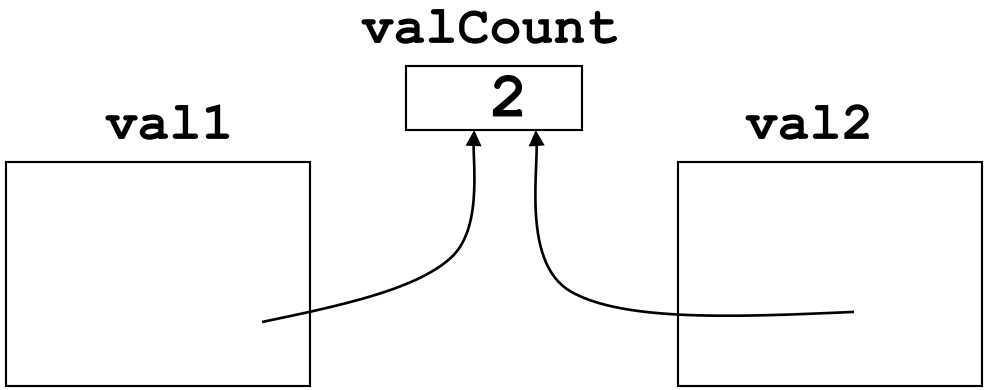
```
class IntVal
{
    //In-class declaration
    static int valCount;
    //Other members not shown
};
//Definition outside of class
int IntVal::valCount = 0;
```



Static Member Variables

3) Can be accessed or modified by any object of the class: Modifications by one object are visible to all objects of the class:

```
IntVal val1, val2;
```



Static Member Functions

1)Declared with **`static`** before return type:

```
class IntVal
{ public:
    static int getValCount()
    { return valCount; }
  private:
    int value;
    static int valCount;
};
```

Static Member Functions

2) Can be called independently of class objects, through the class name:

```
cout << IntVal::getValCount();
```

3) Because of item 2 above, the **`this`** pointer cannot be used

4) Can be called before any objects of the class have been created

5) Used primarily to manipulate static member variables of the class

11.3 Friends of Classes

- Friend function: a function that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with the **`friend`** keyword in the function prototype

Friend Function Declarations

1) Friend function may be a stand-alone function:

```
class aClass
{
  private:
     int x;
     friend void fSet(aClass &c, int a);
};

void fSet(aClass &c, int a)
{
   c.x = a;
}
```

Friend Function Declarations

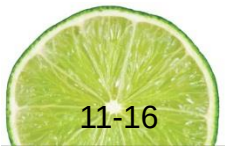
2) Friend function may be a member of another class:

```
class aClass
{ private:
    int x;
    friend void OtherClass::fSet
                          (aClass &c, int a);
};
class OtherClass
{ public:
    void fSet(aClass &c, int a)
    { c.x = a; }
};
```

Friend Class Declaration

3) An entire class can be declared a friend of a class:

```
class aClass
{private:
   int x;
   friend class frClass;
};
class frClass
{public:
   void fSet(aClass &c,int a){c.x = a;}
   int fGet(aClass c){return c.x;}
};
```



Friend Class Declaration

- If **`frClass`** is a friend of **`aClass`**, then all member functions of **`frClass`** have unrestricted access to all members of **`aClass`**, including the private members.

- In general, restrict the property of Friendship to only those functions that must have access to the private members of a class.

11.4 Memberwise Assignment

- Can use **=** to assign one object to another, or to initialize an object with an object's data
- Examples (assuming class `v`):

```
V v1, v2;
… // statements that assign
… // values to members of v1
v2 = v1;    // assignment
V v3 = v2;  // initialization
```

11.5 Copy Constructors

- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor copies field-to-field, using memberwise assignment
- The default copy constructor works fine in most cases

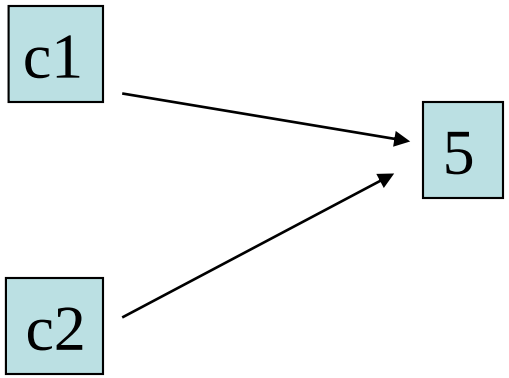
Copy Constructors

Problems occur when objects contain pointers to dynamic storage:

```
class CpClass
{
  private:
    int *p;
  public:
    CpClass(int v=0)
       { p = new int; *p = v;}
    ~CpClass(){delete p;}
};
```

Default Constructor Causes Sharing of Storage

```
CpClass c1(5);
if (true)
{
   CpClass c2=c1;
}
// c1 is corrupted
// when c2 goes
// out of scope and
// its destructor
// executes
```



Programmer-Defined Copy Constructors

- A copy constructor is one that takes a reference parameter to another object of the same class
- The copy constructor uses the data in the object passed as parameter to initialize the object being created
- Reference parameter should be **`const`** to avoid potential for data corruption

Programmer-Defined Copy Constructors

- The copy constructor avoids problems caused by memory sharing
- Can allocate separate memory to hold new object's dynamic member data
- Can make new object's pointer point to this memory
- Copies the data, not the pointer, from the original object to the new object

Copy Constructor Example

```
class CpClass
{
    int *p;
  public:
    CpClass(const CpClass &obj)
    { p = new int; *p = *obj.p; }
    CpClass(int v=0)
    { p = new int; *p = v; }
    ~CpClass(){delete p;}
};
```

Copy Constructor – When Is It Used?

A copy constructor is called when

- An object is initialized from an object of the same class
- An object is passed by value to a function
- An object is returned using a **`return`** statement from a function

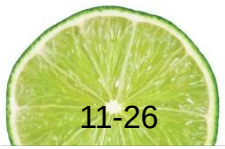
11.6 Operator Overloading

- Operators such as **=**, **+**, and others can be redefined for use with objects of a class

- The name of the function for the overloaded operator is **`operator`** followed by the operator symbol, *e.g.*,

  `operator+` is the overloaded **+** operator and

  `operator=` is the overloaded **=** operator



Operator Overloading

- Operators can be overloaded as
  - instance member functions, or as
  - friend functions
- The overloaded operator must have the same number of parameters as the standard version. For example, `operator=` must have two parameters, since the standard = operator takes two parameters.

Overloading Operators as Instance Members

A binary operator that is overloaded as an instance member needs only one parameter, which represents the operand on the right:

```
class OpClass
{
 private:
    int x;
 public:
    OpClass operator+(OpClass right);
};
```

Overloading Operators as Instance Members

- The left operand of the overloaded binary operator is the calling object
- The implicit left parameter is accessed through the **`this`** pointer

```
OpClass OpClass::operator+(OpClass r)
{   OpClass sum;
    sum.x = this->x + r.x;
    return sum;
}
```

Invoking an Overloaded Operator

- Operator can be invoked as a member function:

```
OpClass a, b, s;
s = a.operator+(b);
```

- It can also be invoked in the more conventional manner:

```
OpClass a, b, s;
s = a + b;
```

Overloading Assignment

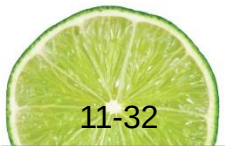
- Overloading the assignment operator solves problems with object assignment when an object contains pointer to dynamic memory.
- Assignment operator is most naturally overloaded as an instance member function
- It needs to return a value of the assigned object to allow cascaded assignments such as

```
a = b = c;
```


Overloading Assignment

Assignment overloaded as a member function:

```
class CpClass
{
    int *p;
  public:
    CpClass(int v=0)
    { p = new int; *p = v;
    ~CpClass(){delete p;}
    CpClass operator=(CpClass);
};
```



Overloading Assignment

Implementation returns a value:

```
CpClass CpClass::operator=(CpClass r)
{
  *p = *r.p;
  return *this;
};
```

Invoking the assignment operator:

```
CpClass a, x(45);
a.operator=(x); // either of these
a = x;          // lines can be used
```

Notes on Overloaded Operators

- Overloading can change the entire meaning of an operator
- Most operators can be overloaded
- Cannot change the number of operands of the operator
- Cannot overload the following operators:

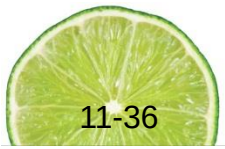
```
?:   .   .*   sizeof
```

Overloading Types of Operators

- **`++`, `--`** operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a **`bool`** value
- Overloaded stream operators **`>>`**, **`<<`** must return **`istream`**, **`ostream`** objects and take **`istream`**, **`ostream`** objects as parameters

Overloaded `[]` Operator

- Can be used to create classes that behave like arrays, providing bounds-checking on subscripts
- Overloaded `[]` returns a reference to object, not an object itself

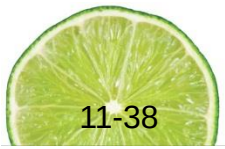


11.7 Type Conversion Operators

- Conversion Operators are member functions that tell the compiler how to convert an object of the class type to a value of another type
- The conversion information provided by the conversion operators is automatically used by the compiler in assignments, initializations, and parameter passing

Syntax of Conversion Operators

- Conversion operator must be a member function of the class you are converting from
- The name of the operator is the name of the type you are converting to
- The operator does not specify a return type



Conversion Operator Example

- To convert from a class **`IntVal`** to an integer:

```
class IntVal
{
   int x;
 public:
   IntVal(int a = 0){x = a;}
   operator int(){return x;}
};
```

- Automatic conversion during assignment:

```
IntVal obj(15); int i;
i = obj;  cout << i; // prints 15
```

11.8 Convert Constructors

Convert constructors are constructors that take a single parameter of a type other than the class in which they are defined

```
class CCClass
{  int x;
  public:
    CCClass()                 //default
    CCClass(int a, int b);
    CCClass(int a);           //convert
    CCClass(string s);  //convert
};
```

Example of a Convert Constructor

The C++ **`string`** class has a convert constructor that converts from C-strings:

```
class string
{
  public:
    string(char *);  //convert
    …
};
```

Uses of Convert Constructors

- They are automatically invoked by the compiler to create an object from the value passed as parameter:

```
string s("hello");   //convert C-string
CCClass obj(24);     //convert int
```

- The compiler allows convert constructors to be invoked with assignment-like notation:

```
string s = "hello"; //convert C-string
CCClass obj = 24;   //convert int
```

Uses of Convert Constructors

- Convert constructors allow functions that take the class type as parameter to take parameters of other types:

```
void myFun(string s); // needs string
                      // object
myFun("hello");       // accepts C-string

void myFun(CCClass c);
myFun(34);            // accepts int
```

Chapter 11: More About Classes and Object-Oriented Programming

Starting Out with C++
Early Objects
Eighth Edition

by Tony Gaddis, Judy Walters, and Godfrey Muganda

Addison-Wesley is an imprint of PEARSON