**Operator Overloading**

Operator overloading is one of the most exciting features of object-oriented programming. It can transform complex, obscure programs into intuitively obvious ones. For example, a statement like:

d3.addObjects(d1, d2);

can be changed to the much more readable:

d3 = d1 + d2;

The term operator overloading refers to giving the normal C++ operators additional meanings when they are applied to user-defined types. We’re already familiar with operator overloading:

c = a + b;

When you use the plus sign with int values, one specific set of machine language instructions is involved. When used with float values, a completely different set of machine language instructions is invoked. Attempting to apply + and = when a, b, and c are objects of a user-defined class will cause complaints from the compiler. However, this statement would become legal with operator overloading. In effect, operator overloading gives us the opportunity to redefine the C++ language.

You can overload the following operators:

• arithmetic +, -, \*, /, %

• logical &&, ||, !

• bitwise &, |, ~, ^, <<, >>

• comparison <, >, <=, >=

• equality ==, !=

• assignment =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=

• comma ,

• increment & decrement ++, --

• subscript []

• function call ()

• class pointer ->

• member pointer selector ->\*

• free store allocator new

• recycling delete

You cannot overload these operators:

• member .

• member object selector .\*

• ternary conditional ?:

• sizeof sizeof

• scope resolution ::

Note the following:

• You can only overload existing operators. That is, you cannot design new operators, tempting as it may be:

:=, <>, |x|, \*\*

• You cannot change operator precedence.

• You cannot change the arity of an operator.

no postfix !

no binary ~

no unary ^

• You cannot define a new action for an intrinsic operator for a native C++ data type. That is, you cannot change the way the + operator works on two ints.

• Overloaded operators may not have default arguments.

To overload an operator, you define a function for the compiler to call when that operator is used with the appropriate data types. Whenever the compiler sees those data types used with the operator, it calls the function.

**Overloading Unary Operators**

Let’s start off by overloading a unary operator. Unary operators act on only one operand (++, --, -).

We’ll look at the Counter class we previously defined. Objects of that class were incremented by calling a member function:

      c1.incCount();

That did the job, but the listing would have been more readable if we could have used the increment operator ++ and simply said:

      ++c1;

Here’s our newest version of the Counter class, rewritten to use an overloaded increment variable:

class Counter {

      private:

            int count;

      public:

            Counter()         { count = 0; }

            int getCount()    { return count; }

            void operator++() { ++count; }

};

void main() {

      Counter c1, c2;

      cout << “c1 = “ << c1.getCount() << endl;

      cout << “c2 = “ << c2.getCount() << endl;

      ++c1;

      ++c2;

      cout << “c1 = “ << c1.getCount() << endl;

      cout << “c2 = “ << c2.getCount() << endl;

}

The keyword operator is used to overload the ++ operator in the class. The overloaded ++ operator function is made up of four components:

           return type                                  void

           keyword operator                         operator

           operator to be overloaded            ++

           argument list in parens                 ()

The compiler calls this member function whenever the ++ operator is encountered, provided the operand is of type Counter.

Note: as seen before, the compiler distinguishes between overloaded functions by the type and number of their arguments. Similarly, the compiler distinguishes between overloaded operators by the types of their operands.

In main(), the ++ operator is applied to a specific object as shown. Yet, operator++() takes no arguments. So what does this operator increment? It increments the count data member in the object c1, the object which used the operator ++. Since member functions can always access the particular object for which they’ve been called, this operator requires no arguments.

The operator++() function has a subtle defect which you’ll see if you to execute this statement:

      c1 = ++c2;

The compiler will complain. Why?

We’ve defined the ++ operator to have a return type of void in the operator++() function. But in the statement:

      c1 = ++c2;

the function is asked to return a value of type Counter. The compiler is being asked to return whatever value c2 has after being operated on by the ++ operator, and then assign this value to c1. So we can’t use our ++ operator to increment Counter objects in expressions.

To make it possible to use our operator++() in expressions, we must provide a way for it to return a value:

class Counter {

      private:

            int count;

      public:

            Counter()         { count = 0; }

            int getCount()    { return count; }

            Counter operator++()

                  {

++count;

Counter temp;

temp.count = count;

return temp;

}

};

In this version of the class, our operator++() function returns a value of class Counter.

           it increments the count data in its own object as before.

           the function creates a new object of type Counter, called temp, to use as a return value

           it assigns count in the temporary object the same value as in itself

           it returns the temp object

These changes now allow for the following style of expression in main():

      c2 = ++ c1;

      ++c1.getCount() // displays itself using the getCount member function.

There are more convenient ways to return objects from functions and overloaded operators. Previously, we created a temporary object to return a function value that took three lines of code. We can actually do it with a simple statement:

Counter operator++() {

      ++count;

      return Counter(count);

}

This code:

           creates an object of type Counter that has no name (it won’t be around long enough to need one)

 this unnamed object is initialized to the value provided by the argument count.

           C++ treats this as a constructor call for a temporary object with no name and a very brief lifetime. However, the code does require a constructor that takes one argument:

            Counter(int c) { count = c; }

           Once the unnamed object is initialized to the value of count, it can then be returned

We can do the same with decrement operators:

Counter operator—() {

      --count;

      return Counter(count);        // or return \*this;

}

**Distinguishing Prefix and Postfix for Operators ++ and --**

In earlier versions of C++, it was impossible to define separate overloaded operations for postfix and prefix ++ and –- operators. However, the compiler can now distinguish between the prefix and postfix versions of these operators by calling the operator function with no arguments or with one int argument, respectively.

class Counter {

      private:

            int count;

      public:

            Counter()         { count = 0; }

            Counter(int x)    { count = x; }

            int getCount()    { return count; }

            Counter operator++() {

++count;

return Counter(count); // or return \*this;

}

            Counter operator++(int) {

                  int temp = count;

                  ++count;

                  return Counter(temp);

            }

};

Member functions operator++(int) and operator--(int) defines a postfix increment and decrement operator for a Counter object. C++ assigns 0 to the single int parameter.

For an object c1 of type Counter, the expression ++c1 calls the overloaded prefix ++ operator, in effect executing the statement:

      c1.operator++();

The expression c1++ calls the overloaded postfix ++ operator and executes as though written:

c1.operator++(0);

The decrement operator works similarily.

Because older versions of C++ did not make the distinction between prefix and postfix, some newer compilers might allow both calls to be made with only the prefix definition of operator++() or operator--(), although they usually issue a warning.

Notice that the argument int has no identifier, so it is not used in the function. It is only there to generate a different function signature. Because there is no identifier, the compiler will not complain that a variable has been “created but not used”.

Our main concern when creating prefix and postfix operators is to decide what incrementing and decrementing means for an object of this class. In most cases, there’s a specific data member in the class that you’ll want to increment or decrement.

**Overloading Binary Operators**

Our previous Distance class was defined as follows:

class Distance {

      private:

            int feet;

            float inches;

      public:

            Distance() {} // default ctor

            Distance (int ft, float in) { feet = ft; inches = in; }

            void addDist(Distance, Distance);

};

void Distance::addDist(Distance d2, Distance d3) {

      feet = d2.feet + d3.feet;

      inches = d2.inches + d3.inches;

      if (inches >= 12.0) {

            inches -= 12.0;

            ++feet;

      }

}

Using this Distance class, we would add two Distance objects as:

      d3.addDist(d1, d2);

We will now overload the + operator to perform the addition of two Distance objects as:

      d3 = d1 + d2;

We can now rewrite the class as:

class Distance {

      private:

            int feet;

            float inches;

      public:

            Distance() {} // default ctor

            Distance (int ft, float in) { feet = ft; inches = in; }

            Distance operator+(Distance d);;

};

void Distance Distance::operator+(Distance d) {

      int ft = feet + d.feet;

      float in = inches + d.inches;

      if (in >= 12.0) {

            in -= 12.0;

            ++ft;

      }

      return Distance(ft, in);

}

When the compiler sees an expression like:

      d3 = d1 + d2;

it realizes it must use the Distance member function operator+(). But what does this function use as its argument, d1 or d2? And why doesn't it need two arguments, since two numbers are added?

The argument on the left side of the operator, d1, is the object that really calls the operator member function. The object on the right side of the operator, d2, must be furnished as an argument to the operator.

In the operator+() function, data members left of the operand are accessed directly as feet and inches, since this is the object which really called the member function. The right operand is accessed as the function's argument, as d2.feet and d2.inches.

Note: Overloaded operators always require one less argument than the number of operands. This is true since one operand is the object which called the member function. That is also why unary operators require no arguments.

Note: If a member function overloads the addition operator, the expression:

      c = a + b;

is symantically equivalent to the expression:

      c = a.operator+(b);

**Overload Arithmetic Assignment Operators (+=)**

This operator combines assignment and addition in one step. We'll use this operator to add one Distance object to a second Distance object, leaving the result in the first. This is similar to the previous Distance program which provided an overloaded + operator with a subtle difference.

void Distance::operator+=(Distance d) {

      feet += d.feet;

      inches += d.inches;

      if (inches >= 12.0) {

            inches -= 12.0;

            ++feet;

      }

}

The previous version of the overloaded + operator returned a Distance object, while here we return nothing. In the previous Distance program we created the overloaded + operator and it allowed us to add two Distance objects and assign the value to a third using:

      d3 = d1 + d2;

This version permits:

      d1 += d2;

The object d2 maps onto d. The object on the left side of the += operator is the object that effectively calls the operator+=() function. Therefore, it has direct access to feet and inches.

Note: Because the function alters d1 directly, adding d2 into it, there is no need to return a value.

Note: If you wanted to use this operator in more complex expressions, like:

      d3 = d1 += d2;

you would need to return a Distance object. This could be done by ending the operator+=() function with a statement such as:

      return Distance(feet, inches);

We would have to change the return value to a Distance object rather than returning a void. This would create a nameless object, initialize it with values, and return it.

**Selecting Friend or Member Functions for Operator Overloading**

In many situations you get equivalent results by using either a friend function or a member function when you overload an operator. A friend function simply contains an extra argument. The friend function must have both objects passed to it, while the member function only needs a single argument. However, sometimes friend functions are too convenient to avoid. One example is when friends are used to increase the versatility of overloaded operators.

For example, the program below shows a limitation in the use of overloaded operators when friends are not used:

#include <iostream.h>

class Distance {

   private:

      int feet;

      float inches;

   public:

      Distance() {                        // default ctor

         feet = 0;

         inches = 0.0;

      }

      Distance(float fFeet) {             // conversion ctor

         feet = int(fFeet);               // feet is integer part

         inches = 12 \* (fFeet – feet);    // inches is what’s left

      }

   Distance (int ft, float in) {

      feet = ft;

      inches = in;

   }

   void showDist() {

      cout << feet << “\’-“ << inches << “\”|;

   }

   Distance operator+(Distance d);

};

void Distance Distance::operator+(Distance d) {

   int ft = feet + d.feet;

   float in = inches + d.inches;

   if (in >= 12.0) {

      in -= 12.0;

      ++ft;

   }

   return Distance(ft, in);

}

void main() {

   Distance d1 = 2.5;

   Distance d2 = 1.25;

   Distance d3;

   cout << “d1 = “ << d1.showDist();

   cout << “d2 = “ << d2.showDist();

   d3 = d1 + 10.0;

   cout << “d3 = “ << d3.showDist();

   d3 = 10.0 + d1;

   cout << “d3 = “ << d3.showDist();

}

Note that the line:

d3 = 10.0 + d1;

produces a compiler error. Why?

In this program, the + operator is overloaded to add two objects of type Distance. There’s also a one-argument constructor that converts a value of type float (representing feet decimal fractions of feet) into a Distance value. That is, converts 10.25 feet into 10 feet and 3 inches. When such a constructor exists, you can make statements like this in main():

d3 = d1 + 10.0;

The overloaded + operator is looking for objects of type Distance both on its left and on its right. But is the argument on the right is type float, the compiler will use the one-argument constructor to convert the float to a Distance value and then carry out the addition.

But why doesn’t:

d3 = 10.0 + d1;

work?

Because the object of which the overloaded + operator is a member must be the variable to the left of the operator. When we place a variable of a different type there, or a constant, there is no object of which the overloaded + operator can be a member and so there is no constructor to convert the type float to type Distance. The compiler simply cannot handle this situation.

A friend function can allow us to write statements that have non-member data types to the left of the operator. So we can change the declaration inside the Distance class to read:

friend Distance operator+(Distance, Distance);

We would now redefine the operator+() function as follows:

Distance operator+(Distance d1, Distance d2) {

   int ft = d1.feet + d2.feet;

   float in = d1.inches + d2.inches;

   if (in >= 12.0) {

      in -= 12.0;

      ++ft;

   }

   return Distance(ft, in);

}

We can now say the following from within main() without a compiler error message:

d3 = d1 + 10.0;

d3 = 10.0 + d1;

Notice that, while the overloaded + operator took one argument as a member function, it takes two as a friend function. In a member function, one of the objects on which a + operates is the object of which it was a member, and the second is an argument. In a friend, both objects must be arguments.

A friend function has one argument for a unary operator and two arguments for a binary operator, while a member function has zero arguments for a unary operator and one argument for a binary operator. This is true because a member function is automatically dealing with one variable already, the object it was called for.

**Concatenating String with + Operator**

Consider the following String class:

const int SIZE = 100;

class String {

   private:

      char buf[SIZE];

   public:

      String() { buf[0] = ‘\0’; }

      String(char s[]) { strcpy(buf, s); }

      void display() { cout << buf; }

      String operator+(String s);

};

String String::operator+(String s) {

   if (strlen(buf) + strlen(s.buf) < SIZE) {

      String temp;

      strcpy(temp.buf, buf);

      strcat(temp.buf, s.buf);

      returnb temp;

   }

   else return String();  // returns NULL String

}

void main() {

   String s1 = “Hello “, s2 = “world!”, s3;

   s3 = s1 + s2;

   s3.display();

}

// output

Hello world!

The + operator takes one argument of type String and returns an object of type String.

Note that with the Counter and Distance classes, we were able to simplify the operator+() function with the statement:

      return Counter(count);

However, here we cannot use:

     return String(String);

to create a nameless temporary String because we need access to the temporary String for several steps. We must be careful that we don’t overflow the fixed-length strings used in the String class. We must check that the combined length of the two strings wil not exceed the maxmum string length. If it does, an overflow error occurs.

Also note that you could put the two overloaded + operator functions together in the same program:

           + operator for String objects

           + operator for Distance objects

and C++ would still know how to interpret the + operator for each context. The compiler would select the correct function to carry out the “addition” based on the type of operand (either Distance or String).

**Overloading the Comparison == Operators**

We can also explore the use of overloaded relational operators to create a comparison operator == to be used with our String class to compare two String objects. The function returns true if the String objects are the same and false is they’re different.

#include <iostream.h>

#include <iomanip.h>

#include <string.h>

const int SIZE = 100;

class String {

   private:

      char buf[SIZE];

   public:

      String() { buf[0] = '\0'; }

      String(char s[]) { strcpy(buf, s); }

      void getString() { cin.get(buf, SIZE); }

      void display() { cout << buf; }

      String operator+(String s);

      int operator==(String s);

};

String String::operator+(String s) {

   if (strlen(buf) + strlen(s.buf) < SIZE) {

      String temp;

      strcpy(temp.buf, buf);

      strcat(temp.buf, s.buf);

      return temp;

   }

   else return String();  // returns NULL String

}

int String::operator==(String s) {

   return !strcmp(buf, s.buf);

}

void main() {

   String s1, s2 = "YES";;

   cout << "Enter YES / NO: ";

   s1.getString();

   if (s2 == s1)

      cout << "You typed YES." << endl;

}

The approach here is similar to overloading the + operator in our previous program with one main difference, the return value. Here we return an int value. Previously we returned an object of class String or an object of class Distance.

**Overloading the Insertion (<<) and Extraction (>>) Operator**

Normally, input and output streams handle only simple data types (int, long, floar, double, char\*, etc). By overloading the iostream input and output stream operators (>>, <<) you can easily add your own classes to the data types that I/O stream statements are designed to use.

For example, we can treat I/O for user-defined data types in the same way as built-in data types. This is a powerful feature of C++. It lets you say:

MyClass foo;

      Cout << foo;

The operator<<() function has already been overloaded for the iostream class, but you can easily overload it for your new class. Lets overload the insertion and extraction operators for the Distance class.

class Distance {

      private:

            int feet;

            float inches;

      public:

            Distance (int ft, float in) { feet = ft; inches = in; }

            friend istream& operator>>(istream& is, Distance& d);

            friend ostream& operator<<(ostream& os, const Distance& d);

};

istream& operator>>(istream& is, Distance& d) {

    cout << “Enter feet: “; is >> d.feet;

    cout << “Enter inches: “; is >> d.inches;

    return is;

}

ostream& operator<<(ostream& os, const Distance& d) {

    os << d.feet << “\’ “ << d.inches << ‘\”’;

    return os;

}

void main() {

   Distance d1, d2;

   cout << “Enter two Distance values: “;

   cin >> d1 >> d2;

   Distance d3(11, 6.25);

   cout << “d1 = “ << d1 << endl;

   cout  << “d2 = “ << d2<< endl;

   cout << “d3 = “ << d3 << endl;

}

This program asks for two distance values from the user, and then prints out these values and another value that was initialized in the program. Note that arguments are passed by reference for efficiency.

The input and output operators require, respectively, an istream and ostream object as their left operand. Both return the object upon which they operate. This allows successive input or output operators to be concatenated.

For example:

((((( cout << “d1 = “) << d1) << “d2 = “) << d2) << endl;)

Here, each parenthetical subexpression returns the ostream object cout which becomes the left operand of the next outermost expression.

Because of the design of iostreams, an overloaded operator<<() or operator>>() function must be a friend function and it must take arguments of an istream object (either istream or ostream, depending on the operator) followed by an object of your user-defined type. The function must return the same stream object as it takes as an argument.

Note, it’s important that the stream object be passed into and out of the function so you can have expressions of the form:

cout << obj1 << obj2

In effect, each object is passed to the stream and then the stream is passed down the line.

Note, why couldn’t we define the overloaded operator function as a member function of the Distance class? If we did, here would be the declaration of the insertion output operator as a member function of Distance:

class Distance {

   …

   ostream& operator<<(ostream&);

  …

};

The left operand of every member function is an object or pointer to an object of its class. This is why the member function instance of the output operator declares only the one ostream object. A call of this instance takes the following form:

d << cout;

It would be very confusing both to the programmer and the human readers of the program to provide this instance. Therefore it’s better to use friend functions.

**Updated Distance Class**

**// dist4.h**

#ifndef \_DIST4\_H

#define \_DIST4\_H

#include <iostream.h>

class Distance {

   private:

      int feet;

      float inches;

      void normalize();

   public:

      Distance();

      Distance(float f);

      Distance(int ft, float in);

      Distance operator+=(const Distance&);

      friend Distance operator+(const Distance&, const Distance&);

      friend istream& operator>>(istream&, Distance&);

      friend ostream& operator<<(ostream&, const Distance&);

};

#endif

**// dist4.cpp**

#include <iostream.h>

#include "dist4.h"

Distance::Distance() {

     feet = 0;

     inches = 0.0;

}

Distance::Distance(float f) {

     feet = int (f);

     inches = 12 \* (f - feet);

}

Distance::Distance(int ft, float in) {

     feet = ft;

     inches = in;

     normalize();

}

void Distance::normalize() {

     while(inches >= 12.0) {

        inches -= 12.0;

        feet++;

     }

}

Distance Distance::operator+=(const Distance& d) {

     feet += d.feet;

     inches += d.inches;

     return Distance(feet, inches);

}

Distance operator+(const Distance& d1, const Distance& d2) {

     int ft = d1.feet + d2.feet;

     float in = d1.inches + d2.inches;

     return Distance(ft, in);

}

istream& operator>>(istream& is, Distance& d) {

     cout << "Enter feet: ";   is >> d.feet;

     cout << "Enter inches: "; is >> d.inches;

     d.normalize();

     return is;

}

ostream& operator<<(ostream& os, const Distance& d) {

     os << d.feet << "\' " << d.inches << "\"";

     return os;

}

**// distdrv4.cpp**

#include <iostream>

#include "dist4.h"

Using namespace std;

void main() {

     Distance d1(5, 22), d2(8.25), d3, d4, d5;

     cout << "d1: " << d1 << endl;

     cout << "d2: " << d2 << endl;

     cout << "d3: " << d3 << endl;

     cout << "d4: " << d4 << endl;

     cout << "d5: " << d5 << endl;

     d3 = d1 + d2;

     cout << "d3 = d1 + d2: " << d3 << endl;

     d4 = d3 + 10;

     cout << "d4 = d3 + 10: " << d4 << endl;

     cin >> d5;

     cout << "d5: " << d5 << endl;

}

**// Output**

d1: 6' 10"

d2: 8' 3"

d3: 0' 0"

d4: 0' 0"

d5: 0' 0"

d3 = d1 + d2: 15' 1"

d4 = d3 + 10: 25' 1"

Enter feet: 8

Enter inches: 31

d5: 10' 7"

**Overloading the Array Subscript Operator []**

The subscript operator [] is normally used to index arrays. In fact, the subscript operator is a binary operator which requires two arguments. For example:

P = x[i];

[]  operator

x  first argument

i  second argument

The subscript operator actually performs a useful function, it hides pointer arithmetic for us! For example, if we have the following array:

char name[20];

and we execute a statement such as:

ch = name[12];

the [] operator directs the assignment statement to add 12 to the base address of the array name to locate the data stored in this memory location.

ch = name[12]  \*(name + 12)

You can overload the array subscript operator [] to provide array-like access to a class’s data members, even though that data might be stored as individual members.

Lets overload the [] operator for a class that stores four integer values as separate data members.

**// parray.h**

#ifndef \_PARRAY\_H

#define \_PARRAY\_H

class PseudoArray {

  private:

     int value0, value1, value2, value3;

  public:

     PseudoArray(int v0, int v1, int v2, int v3);

     int getInt(int i);

     int operator[](int i);

};

#endif

**// parray.cpp**

#include "parray.h"

PseudoArray::PseudoArray(int v0, int v1, int v2, int v3) {

     value0 = v0;

     value1 = v1;

     value2 = v2;

     value3 = v3;

}

int PseudoArray::getInt(int i) {

     switch(i) {

        case 0 : return value0;

        case 1 : return value1;

        case 2 : return value2;

        case 3 : return value3;

        default: return value0;

     }

}

int PseudoArray::operator[](int i) {

     return getInt(i);

}

**// parraydr.cpp**

#include <iostream>

#include "parray.h"

using namespace std;

void main() {

     PseudoArray pa(10, 20, 30, 40);

     for (int i = 0; i < 4; i++)

        cout << "pa[" << i << "] == " << pa[i] << endl;

}

**// parray output**

pa[0] == 10

pa[1] == 20

pa[2] == 30

pa[3] == 40

To get a data element we might have previously said:

cout << pa.getInt(i);

We can now say:

cout << pa[I];

We are using object pa as if it were an array! Overloading the subscript operator makes it possible to use array indexing rather than call member functions for an object of a class.

The overloaded subscript operator function can only be a class member:

pa[I]        pa.operator[](i)

Note that only one explicit argument is needed because the this pointer is provided.

**Another String Example**

Suppose we have a String class that represents strings of different lengths and we wish to provide bounds-checking for array subscripting operations on the strings.

In our example, we’ll support strings that are statically allocated and can hold a maximum of 255 characters:

class String255 {

   private:

      char buf[255];          // maximum size

      int size;               // actual size

   public:

      String255(int num) { size = num; }

      char& operator[](int I);

};

char& String255::operator[](int i) {

   if (i < 0 || i >= SIZE) {

      cout << “subscript out of bounds” << endl;

      exit(1);

   }

   else

      return buf[i];          // the normal [] operator used here

}

void main() {

   String255 a(10);

   char c;

   a[5] = 17;                 // a.operator[](5) = 17;

   c = a[7];                  // c = a.operator[](7);

   c = a[26];                 // c = a.operator[](26);      error

}

In each case, the hidden first operand is a pointer to the object being subscripted, the object a.

Note that the operator[]() function passes back a reference to a character. We do this so that we can use it on both sides of an assignment statement.

Note also that the subscript operator must be able to appear on both the right and the left hand side of an expression. In order to appear on the left-hand side, its return value must be an lvalue. This is achieved by specifying the return value as a reference type:

char& String255::operator[](int i) {

   return buf[I];

}

The return value of the subscript operator is the lvalue of the indexed element. That’s why is can appear as the target off an assignment.

C++ has reference declarations, and such type modifiers produce lvalue’s (stands for location value).

Note: on the right side of an assignment expression, an lvalue is automatically dereferenced. On the left side of an expression, an lvalue specifies where an appropriate value is to be stored.

**One Last Example**

**// phone.h**

#ifndef \_PHONE\_H

#define \_PHONE\_H

typedef struct {

     char \*name;

     long number;

} ENTRY;

class PhoneBook {

   private:

      int length;

      int count;

      ENTRY \*listing;

   public:

      PhoneBook(int);

      ~PhoneBook();

      void addEntry(char\*, long);

      void displayPhoneBook();

      long operator[](char \*);

      char \*operator[](long);

};

#endif

**// phone.cpp**

#include <iostream>

#include <iomanip>

#include <string.h>

#include "phone.h"

using namespace.std;

PhoneBook::PhoneBook(int size) {   // CREATE PHONE BOOK of a specified size

     listing = new ENTRY[size];

     length = size;

     count = 0;

}

PhoneBook::~PhoneBook() {

     for (int i = 0; i < count; i++)

        delete [] listing[i].name;

     delete [] listing;

}

void PhoneBook::displayPhoneBook() {

     for (int i = 0; i < count; i++)

        cout << setw(10) << listing[i].name

             << setw(10) << listing[i].number << endl;

}

void PhoneBook::addEntry(char \*Name, long Number) {

     listing[count].name = new char[strlen(Name) + 1];

     strcpy(listing[count].name, Name);

     listing[count++].number = Number;

}

long PhoneBook::operator[](char \*Name) {  // look up persons number

     for (int i = 0; i < count; i++)

        if (strcmp(listing[i].name, Name) == 0)

             return (listing[i].number);

     return 0;

}

char \*PhoneBook::operator[](long Number) {      // look up persons name

     for (int i = 0; i < count; i++)

        if (listing[i].number == Number)

             return listing[i].name;

     return "Person not found.";

}

**// pbook.cpp**

#include <iostream>

#include "phone.h"

using namespace std;

void main() {

       PhoneBook pb(100);

       long number;

       char \*name;

       pb.addEntry("Haley"   , 4861144);

       pb.addEntry("Shari"   , 4933684);

       pb.addEntry("Samantha", 4861199);

       cout << "Display phone book:" << endl << endl;

       pb.displayPhoneBook();

       cout << endl;

       number = pb["Shari"];

       cout << "number = " << number << endl;

       name = pb[4861199];

       cout << "  name = " << name << endl;

}