3. Copy Constructor

One of the more important forms of an overloaded constructor is the *copy constructor*. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another.

By default, when one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object. There are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created.

For example, assume a class called *MyClass* that allocates memory for each object when it is created, and an object *A* of that class. This means that *A* has already allocated its memory. Further, assume that *A* is used to initialize *B*, as shown here:

MyClass B = A;

If a bitwise copy is performed, then *B* will be an exact copy of *A*. This means that *B* will be using the same piece of allocated memory that *A* is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if *MyClass* includes a destructor that frees the memory, then the same piece of memory will be freed twice when *A* and *B* are destroyed!

The same type of problem can occur in two additional ways:

1. When a copy of an object is made when it is passed as an argument to a function;
2. When a temporary object is created as a return value from a function.

To solve these types of problems, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. Thus, your copy constructor bypasses the default bitwise copy.

General form:

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

Here, *o* is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them. However, in all cases the first parameter must be a reference to the object doing the initializing.

PROGRAM 19: COPY CONSTRUCTOR

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

class Samplecopyconstructor
{
   private:
   int x, y;   //data members

   public:
   Samplecopyconstructor(int x1, int y1)
   {
      x = x1;
      y = y1;
```

```
    }
    /* Copy constructor */
    Samplecopyconstructor (const Samplecopyconstructor &sam)
    {
       x = sam.x;
       y = sam.y;
    }

    void display()
    {
       cout<<x<<" "<<y<<endl;
    }
};
/* main function */
int main()
{
    Samplecopyconstructor obj1(10, 15);     // Normal constructor
    Samplecopyconstructor obj2 = obj1;      // Copy constructor
    cout<<"Normal constructor : ";
    obj1.display();
    cout<<"Copy constructor : ";
    obj2.display();
    return 0;
}
```

OUTPUT:

```
Normal constructor : 10 15
Copy constructor : 10 15
```

DESTRUCTOR

The complement of the constructor is the *destructor*. In many circumstances, an object will need to perform some action or actions when it is destroyed.

Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor (if it has one) is automatically called. There are many reasons why a destructor may be needed. For example, an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened.

In C++, it is the destructor that handles deactivation events. The destructor has the same name as the constructor, but it is preceded by a ~.

PROGRAM 20: DESTRUCTORS

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

class myclass
{
public:
  int who;
```

```
   myclass (int id);
  ~myclass ();
} glob_ob1 (1), glob_ob2 (2);
myclass::myclass (int id)
{
 cout << "Initializing " << id << "\n";
 who = id;
}

myclass::~myclass ()
{
 cout << "Destructing " << who << "\n";
}

int main ()
{
 myclass local_ob1 (3);
 cout << "This will not be first line displayed.\n";
 myclass local_ob2 (4);
 return 0;
}
```

OUTPUT:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1
```

PROGRAM 21: DESTRUCTORS

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

class A
{
  public:
   // constructor
   A()
   {
      cout << "Constructor called"<<endl;
   }

   // destructor
   ~A()
```

```
    {
        cout << "Destructor called"<<endl;
    }
};

int main()
{
    A obj1;   // Constructor Called
    int x = 1;
    if(x)
    {
        A obj2;  // Constructor Called
    }   // Destructor Called for obj2
} //  Destructor called for obj1
```

OUTPUT:

```
Constructor called
Constructor called
Destructor called
Destructor called
```

CONST MEMBER FUNCTION

Class member functions may be declared as const, which causes this to be treated as a const pointer. Thus, that function cannot modify the object that invokes it. Also, a const object may not invoke a non-const member function. However, a const member function can be called by either const or non-const objects.

To specify a member function as const, use the form shown in the following example.

```
class X
{
        int some_var;
        public:
        int f1() const; // const member function
};
```

Purpose

The purpose of declaring a member function as const is to prevent it from modifying the object that invokes it.

Mutable

Sometimes there will be one or more members of a class that you want a const function to be able to modify even though you don't want the function to be able to modify any of its other members. You can accomplish this through the use of mutable. It overrides constness. That is, a mutable member can be modified by a const member function.

PROGRAM 12: CONST MEMBER FUNCTION WITH MUTABLE

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

class Demo
```

```
{
  mutable int i;
  int j;
  public:
  int geti () const
  {
     return i;                    // ok
  }
  void seti (int x) const
  {
    i = x;                        // now, OK.
  }
/* The following function won't compile.
void setj(int x) const
 {
    j = x; // Still Wrong!
 }
*/
};
int main ()
{

Demo ob;
ob.seti (1900);
cout << ob.geti ();
return 0;
}
```

i is specified as mutable, so it may be changed by the seti( ) function. However, j is not mutable and setj( ) is unable to modify its value.

OUTPUT:

```
1900
```

THE SCOPE RESOLUTION OPERATOR

The primary use of **::** operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name.

For example, consider this fragment:

```
int i; // global i
void f()
{
        int i; // local i
        i = 10; // uses local i
        ...
        ...
        ...
}
```

The assignment **i = 10** refers to the local **i**. But what if function **f( )** needs to access the global version of **i**? It may do so by preceding the **i** with the **::** operator, as shown here.

```
int i; // global i
void f()
{
        int i; // local i
        ::i = 10; // now refers to global i
        ...
        ...
        ...
}
```

PROGRAM 22: SCOPE RESOLUTION OPERATOR: LAB PROGRAM WEK -7

Arrays of Objects

In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for any other type of array.

PROGRAM 23: ARRAYS OF OBJECTS WITHOUT INITIALIZATION OF OBJECTS

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

class cl
{
  int i;
  public:
  void set_i(int j)
   {
        i=j;
    }
  int get_i()
   {
        return i;
   }
};
int main()
{
   cl ob[3];
   int i;
   for(i=0; i<3; i++)
   ob[i].set_i(i+1);
   for(i=0; i<3; i++)
   cout << ob[i].get_i() << "\n";
   return 0;
}
```

OUTPUT:

```
1
```

```
2
3
```

Initializing objects

If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructors.

1. **Only one parameter**

   For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax. This is a short form. As each element in the array is created, a value from the list is passed to the constructor's parameter.

PROGRAM 24: ARRAYS OF OBJECTS WITH INITIALIZATION OF OBJECTS WITH ONE ARGUMENT

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

class cl
{
   int i;
   public:
   cl(int j)
   {
         i=j;
   } // constructor
  int get_i()
  {
      return i;
  }
};
int main()
{
   cl ob[3] = {1, 2, 3}; // initializers
   int i;
   for(i=0; i<3; i++)
   cout << ob[i].get_i() << "\n";
   return 0;
}
```

OUTPUT:

```
1
2
3
```

2. Two or more arguments

If an object's constructor requires two or more arguments, you will have to use the longer initialization form.

PROGRAM 25: ARRAYS OF OBJECTS WITH INITIALIZATION OF OBJECTS WITH TWO OR MORE ARGUMENTS

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

 class cl
{
 int h;
 int i;
public:
cl (int j, int k)
  {
    h = j;
    i = k;
  }                                  // constructor with 2 parameters
  int get_i ()
  {
    return i;
  }

int get_h ()
  {
    return h;
  }
};

int main ()
{
cl ob[3] =
  {
      cl (1, 2),                     // initialize
      cl (3, 4),
      cl (5, 6)
};
int i;
for (i = 0; i < 3; i++)
{
   cout << ob[i].get_h ();
   cout << ", ";
   cout << ob[i].get_i () << "\n";
}
return 0;
}
```

OUTPUT:

```
1, 2
3, 4
5, 6
```

PROGRAM 26: ARRAY OF CLASS OBJECTS LAB PROGRAM WEEK -1

Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator.

PROGRAM 27: POINTERS TO OBJECTS

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

class cl
{
  int i;
  public:
  cl (int j)
 {
     i = j;
 }
  int get_i ()
 {
   return i;
 }
};

int main ()
{
   cl ob (88), *p;
   p = &ob;                    // get address of ob
   cout << p->get_i ();        // use -> to call get_i()
  return 0;
}
```

OUTPUT:

```
88
```

When a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer. The same is true of pointers to objects.

PROGRAM 28: POINTER INCREMENT

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

class cl
{
```

```
    int i;
    public:
   cl ()
   {
      i = 0;
   }
  cl (int j)
  {
     i = j;
  }
int get_i ()
  {
    return i;
  }
};

int main ()
{
  cl ob[3] =
  {
  1, 2, 3};
  cl * p;
  int i;
  p = ob;                              // get start of array
  for (i = 0; i < 3; i++)
    {
  cout << p->get_i () << "\n";
  p++;                          // point to next object
    }
  return 0;
}
```

OUTPUT:

```
1
2
```

We can assign the address of a public member of an object to a pointer and then access that member by using the pointer.

PROGRAM 29: ASSIGN THE ADDRESS OF A PUBLIC MEMBER OF AN OBJECT TO A POINTER AND THEN ACCESS THAT MEMBER

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

class cl
{
   public:
   int i;
  cl (int j)
```

```
  {
    i = j;
   }
};


int main ()
{
  cl ob (1);
  int *p;
  p = &ob.i;                    // get address of ob.i
  cout << *p;                   // access ob.i via p
  return 0;
}
```

OUTPUT:

```
1
```

PROGRAM 30: POINTER TO CLASS LAB PROGRAM WEEK-2

THE this POINTER

When defining member functions for a class, you sometimes want to refer to the calling object. The this pointer is a predefined pointer that points to the calling object.

PROGRAM 31: THIS POINTER

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

 class pwr
{
 double b;
 int e;
 double val;
 public:
 pwr (double base, int exp);
 double get_pwr ()
  {
    return val;
  }
};
pwr::pwr (double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if (exp == 0)
    return;
  for (; exp > 0; exp--)
```

```
    val = val * b;
}

int main ()
{
   pwr x (4.0, 2), y (2.5, 1), z (5.7, 0);
cout << x.get_pwr () << " ";
cout << y.get_pwr () << " ";
cout << z.get_pwr () << "\n";
return 0;
}
```

OUTPUT:

```
16 2.5 1
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside **pwr( )**, the statement

b = base;

means that the copy of **b** associated with the invoking object will be assigned the value contained in **base**. However, the same statement can also be written like this:

this->b = base;

The **this** pointer points to the object that invoked **pwr( )**. Thus, **this –>b** refers to that object's copy of **b**. For example, if **pwr( )** had been invoked by **x** (as in **x(4.0, 2)**), then **this** in the preceding statement would have been pointing to **x**. Writing the statement without using **this** is really just shorthand.

The **this** pointer is automatically passed to all member functions.Therefore, **get_pwr( )** could also be rewritten as shown here:

double get_pwr() { return this->val; }

In this case, if **get_pwr( )** is invoked like this:

y.get_pwr();

then **this** will point to object **y**.

Two important points about this.

1. **Friend** functions are not members of a class and, therefore, are not passed a **this** pointer.
2. **Static** member functions do not have a **this** pointer.

PROGRAM 32: THIS POINTER

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

class Box
{
  public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0)
   {
      cout <<"Constructor called." << endl;
```

```
      length = l;
      breadth = b;
      height = h;
    }
    double Volume() {
      return length * breadth * height;
    }
    int compare(Box box) {
      return this->Volume() > box.Volume();
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};
int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  if(Box1.compare(Box2))
 {
    cout << "Box2 is smaller than Box1" <<endl;
  }
 else
 {
    cout << "Box2 is equal to or larger than Box1" <<endl;
  }
    return 0;
}
```

OUTPUT:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

When local variable's name is same as member variables name then this pointer can be used resolve the name clashes.

PROGRAM: THIS POINTER

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

/* local variable is same as a member's name */
class Test
{
private:
  int x;
public:
```

```
  void setX (int x)
  {
     // The 'this' pointer is used to retrieve the object's x
     // hidden by the local variable 'x'
     this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

OUTPUT:

```
x = 20
```

Pointers to Class Members

C++ allows you to generate a special type of pointer that "points" generically to a member of a class, not to a specific instance of that member in an object. This sort of pointer is called a *pointer to a class member* or a *pointer-to-member,* for short.

A pointer to a member is not the same as a normal C++ pointer. Instead, a pointer to a member provides only an offset into an object of the member's class at which that member can be found. Since member pointers are not true pointers, the **.** and **–>** cannot be applied to them.

To access a member of a class given a pointer to it, you must use the special pointer-to-member operators **.*** and **–>***. Their job is to allow you to access a member of a class given a pointer to that member.

PROGRAM 33: POINTERS TO CLASS MEMBERS (ACCESSING A MEMBER OF AN OBJECT BY USING AN OBJECT USING USE THE .* OPERATOR)

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

class cl
{
public:
cl (int i)
  {
    val = i;
  }
int val;

int double_val ()
  {
```

```
    return val + val;
  }
};

int main ()
{
  int cl::*data;              // data member pointer
  int (cl::*func) ();         // function member pointer
  cl ob1 (1), ob2 (2);        // create objects
  data = &cl::val;            // get offset of val
  func = &cl::double_val;     // get offset of double_val()
  cout << "Here are values: ";

cout << ob1.*data << " " << ob2.*data << "\n";
cout << "Here they are doubled: ";
cout << (ob1.*func) () << " ";
cout << (ob2.*func) () << "\n";
return 0;
}
```

OUTPUT:

```
Here are values: 1 2
Here they are doubled: 2 4
```

When you are accessing a member of an object by using an object or a reference, you must use the **.*** operator. However, if you are using a pointer to the object, you need to use the **–>*** operator.

PROGRAM 34: POINTERS TO CLASS MEMBERS (USING A POINTER TO THE OBJECT, YOU NEED TO USE THE –>* OPERATOR)

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

class cl
{
public:
  cl (int i)
  {
    val = i;
  }
  int val;
  int double_val ()
  {
    return val + val;
  }
};

int main ()
{
```

```
  int cl::*data;            // data member pointer
  int (cl::*func) ();               // function member pointer
  cl ob1 (1), ob2 (2);              // create objects
  cl *p1, *p2;
  p1 = &ob1;                        // access objects through a pointer
  p2 = &ob2;
  data = &cl::val;                  // get offset of val
  func = &cl::double_val;           // get offset of double_val()
  cout << "Here are values: ";
  cout << p1->*data << " " << p2->*data << "\n";
  cout << "Here they are doubled: ";
  cout << (p1->*func) () << " ";
  cout << (p2->*func) () << "\n";
  return 0;
}
```

OUTPUT:

```
Here are values: 1 2
Here they are doubled: 2 4
```

Inline Function in classes

Inline functions may be class member functions.

PROGRAM: INLINE FUNCTION

```
#include <iostream>
using namespace std;
class myclass
{
        int a, b;
        public:
        void init(int i, int j);
        void show();
};
// Create an inline function.
inline void myclass::init(int i, int j)
{
        a = i;
        b = j;
}
// Create another inline function.
inline void myclass::show()
{
        cout << a << " " << b << "\n";
}
int main()
{
        myclass x;
        x.init(10, 20);
        x.show();
        return 0;
```

```
}
```

Output:

```
10 20
```

*The **inline** keyword is not part of the C subset of C++. Thus, it is not defined by C89. However, it has been added by C99.*

Defining Inline Functions within a Class

It is possible to define short functions completely within a class declaration. When a function is defined inside a class declaration, it is automatically made into an **inline** function (if possible). It is not necessary (but not an error) to precede its declaration with the **inline** keyword.

For example, the preceding program is rewritten here with the definitions of **init( )** and **show( )** contained within the declaration of **myclass**:

```
#include <iostream>
using namespace std;
class myclass
{
        int a, b;
        public:
        // automatic inline
        void init(int i, int j)
        {
                a=i; b=j;
        }
        void show()
        {
                cout << a << " " << b << "\n";
        }
};
int main()
{
        myclass x;
        x.init(10, 20);
        x.show();
        return 0;
}
```

Constructor and destructor functions may also be inlined, either by default, if defined within their class, or explicitly.

Passing References to Objects

When an object is passed as an argument to a function, a copy of that object is made. When the function terminates, the copy's destructor is called. However, when you pass by reference, no copy of the object is made. This means that no object used as a parameter is destroyed when the function terminates, and the parameter's destructor is not called.

PROGRAM 71: PASSING REFERENCES TO OBJECTS

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

```
class cl {
int id;
public:
int i;
cl(int i);
~cl();
void neg(cl &o) { o.i = -o.i; } // no temporary created
};
cl::cl(int num)
{
cout << "Constructing " << num << "\n";
id = num;
}
cl::~cl()
{
cout << "Destructing " << id << "\n";
}
int main()
{
cl o(1);
o.i = 10;
o.neg(o);
cout << o.i << "\n";
return 0;
}
```

OUTPUT:

```
Constructing 1
-10
Destructing 1
```

Passing Objects to Functions

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by value mechanism.

PROGRAM 35: PASSING OBJECTS TO FUNCTIONS

```
// Passing an object to a function.
#include <iostream>
using namespace std;

class myclass
{
   int i;
   public:
   myclass (int n);
  ~myclass ();
   void set_i (int n)
  {
    i = n;
```

```
  }
   int get_i ()
  {
    return i;
  }
};
myclass::myclass (int n)
{
  i = n;
  cout << "Constructing " << i << "\n";
}
myclass::~myclass ()
{
  cout << "Destroying " << i << "\n";
}
void f (myclass ob);

int main ()
{
  myclass o (1);
  f (o);

cout << "This is i in main: ";
cout << o.get_i () << "\n";
return 0;
}
void f (myclass ob)
{
  ob.set_i (2);
  cout << "This is local i: " << ob.get_i ();
  cout << "\n";
}
```

OUTPUT:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

When a copy of an object is created to be used as an argument to a function, the normal constructor is not called. Instead, the default copy constructor makes a bit-by-bit identical copy. However, when the copy is destroyed (usually by going out of scope when the function returns), the destructor is called. Because the default copy constructor creates an exact duplicate of the original, it can, at times, be a source of trouble. Even though objects are passed to functions by means of the normal call-by-value parameter passing mechanism which, in theory, protects and insulates the calling argument, it is still possible for a side effect to occur that may affect, or even damage, the object used as an argument. For example,

if an object used as an argument allocates memory and frees that memory when it is destroyed, then its local copy inside the function will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless. To prevent this type of problem you will need to define the copy operation by creating a copy constructor for the class.

Returning Objects

A function may return an object to the caller.

PROGRAM 36: RETURNING OBJECTS FROM A FUNCTION

```
// Returning objects from a function.
#include <iostream>
using namespace std;

 class myclass
{
  int i;
  public:
  void set_i (int n)
  {
    i = n;
  }
  int get_i ()
  {
    return i;
  }
};

myclass f ();                    // return object of type myclass
int main ()
{
  myclass o;
  o = f ();
  cout << o.get_i () << "\n";
  return 0;
}
myclass f ()
{
  myclass x;
  x.set_i (1);
  return x;
}
```

OUTPUT:

```
1
```

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object

may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it. There are ways to overcome this problem that involve overloading the assignment operator and defining a copy constructor.

Object Assignment

Assuming that both objects are of the same type, you can assign one object to another. This causes the data of the object on the right side to be copied into the data of the object on the left. For example, this program displays **99**:

PROGRAM: OBJECT ASSIGNMENT

```
// Assigning objects.
#include <iostream>
using namespace std;
class myclass
{
        int i;
        public:
        void set_i(int n)
        {
                i=n;
         }
        int get_i()
        {
                return i;
        }
};
int main()
{
        myclass ob1, ob2;
        ob1.set_i(99);
        ob2 = ob1; // assign data from ob1 to ob2
        cout << "This is ob2's i: " << ob2.get_i();
        return 0;
}
```

By default, all data from one object is assigned to the other by use of a bit-by-bit copy. However, it is possible to overload the assignment operator and define some other assignment procedure.

DYNAMICALLY ALLOCATING AND DEALLOCATING OBJECTS

You can allocate objects dynamically by using **new**. When you do this, an object is created and a pointer is returned to it. The dynamically created object acts just like any other object. When it is created, its constructor (if it has one) is called. When the object is freed, its destructor is executed.

PROGRAM 76: DYNAMIC MEMORY ALLOCATION AND DEALLOCATION OF OBJECTS

```
#include <iostream>
#include<string.h>
```

```
using namespace std;

class balance
{
	double cur_bal;
	char name[80];
	public:
	void set(double n, char *s)
	{
		cur_bal = n;
		strcpy(name, s);
	}
	void get_bal(double &n, char *s)
	{
		n = cur_bal;
		strcpy(s, name);
	}
};
int main()
{
	balance *p;
	char s[80];
	double n;
	try
	{
		p = new balance;
	}
	catch (bad_alloc xa)
	{
		cout << "Allocation Failure\n";
		return 1;
	}
	p->set(12387.87, "Ralph Wilson");
	p->get_bal(n, s);
	cout << s << "'s balance is: " << n;
	cout << "\n";
	delete p;
	return 0;
}
```

OUTPUT:

```
Ralph Wilson's balance is: 12387.9
```

As stated, dynamically allocated objects may have constructors and destructors. Also, the constructors can be parameterized.

```
#include <iostream>
#include <new>
```

```
#include <cstring>
using namespace std;
class balance
{
	double cur_bal;
	char name[80];
	public:
	balance(double n, char *s)
	{
		cur_bal = n;
		strcpy(name, s);
	}
	~balance()
	{
		cout << "Destructing ";
		cout << name << "\n";
	}
	void get_bal(double &n, char *s)
	{
		n = cur_bal;
		strcpy(s, name);
	}
};
int main()
{
	balance *p;
	char s[80];
	double n;
	// this version uses an initializer
	try
	{
		p = new balance (12387.87, "Ralph Wilson");
	}
	catch (bad_alloc xa)
	{
		cout << "Allocation Failure\n";
		return 1;
	}
	p->get_bal(n, s);
	cout << s << "'s balance is: " << n;
	cout << "\n";
	delete p;
	return 0;
}
```