# Classes and Objects
## Classes and Structures

- Classes and structures - ways of grouping variables of different types
  - similar to records in other languages
  - a C++ class is a generalization of a structure in C
- C++ has both classes and structures

# Classes and Objects
## Classes and Structures

- Differences between Classes and C structures
  - Encapsulation
    - » classes (and C++ structures) can have functions as their members
    - » these functions operate on the data members
  - Data hiding
    - » classes provide form member-access control
    - » for each component in a class or structure we can indicate whether or not data hiding is to apply
    - » defaults:
      - class: full data-hiding
      - structure: no data-hiding

# Classes and Objects
## Classes and Structures

- A class is a type
  - a variable of such a type is called an object
  - (more specifically, a class object)
- Members of objects are accessed via the member names

# Classes and Objects
## Classes and Structures

- Example of Structure Declaration

```
struct article {
   int code;
   char name[20];
   float weight, length;
};
```

– this structure is a class with only public members

# Classes and Objects
## Classes and Structures

- Equivalent example of Class Declaration

```
class article {
public:
   int code;
   char name[20];
   float weight, length;
};
```

– this structure is a class with only public members

# Classes and Objects
## Classes and Structures

- Declaration (and definition) of a class variable (i.e. object)

```
article s, t;  // in C we would have
               // had to write
               // struct article s, t;
```

  - This form is identical to the conventional variable definitions

# Classes and Objects
## Classes and Structures

- Since class declarations will probably be used by many program modules, it is good practice to put them in a header file which can be included as required

```
#include “my_classes.h”
```

# Classes and Objects
## Classes and Structures

- Access to members of class variables

```
s.code = 123;
strcpy(s.name, “Pencil”);
s.weight = 12.3;
s.length = 150.7;
t = s;  // !! object assignment possible
        // even though a member is an
        // array (and array assignment
        // is not possible)
```

# Classes and Objects
## Classes and Structures

- Composite variables such as classes, structures, arrays are called aggregates
- Members of aggregrates which are in turn aggregates are called subaggregates

# Classes and Objects
## Classes and Structures

- Pointers to class objects

```
article *p;
...
(*p).code = 123; // normal dereferncing
p->code = 123;   // more usual shorthand
```

# Classes and Objects
## Classes and Structures

- Initialization of class objects

```
class article {
public:
   int code;
   char name[20;
   float weight, length;
};
int main()
{ static article s = (246, “Pen”, 20.6, 147.0},
  t;  // is t initialized?
  ...
```

# Classes and Objects
## Classes and Structures

- Initialization of class objects with object of identical type

```
...
article s = (246, “Pen”, 20.6, 147.0};
...

void f(void)
{ article t=s, u=t;
  ...
}
```

# Classes and Objects
## Classes and Structures

- Subaggregates
  - access to array members

```
s.name[3] = ‘A’; // i.e. (s.name)[3]
```

  - arrays of class objects

```
article table[2];
...
table[i].length = 0;
table[i].name[j] = ‘ ‘;
```

# Classes and Objects
## Classes as Arguments and Return Values

- Three possibilities
  - By ‘value’; entire class object as argument/return value
  - By address
  - By reference (effectively the same as address)

- The following three examples will use a show how a function can create a new object with values based on an object passed to it

# Classes and Objects
## Classes as Arguments and Return Values

- Assume the following class declaration

```
// ARTICLE.H: header file for 3 demos
class article {
public:
   int code;
   char name[20];
   float weight, length;
};
```

# Classes and Objects
## Classes as Arguments and Return Values

- Pass by value: copy class objects

```
// ENTOBJ: passing an entire class object
#include <iostream.h>
#include “article.h”

article largeobj(article x)   // functional spec
{  x.code++;                  // increment code
   x.weight *=2 ;             // double weight
   x.length *=2;              // double length
   return x;                  // return new obj.
}
// main to follow
```

# Classes and Objects
## Classes as Arguments and Return Values

- Pass by value: copy class objects

```
int main()
{  article s = (246, “Pen”, 20.6, 147.0), t;
   t = largeobj(s);
   cout << t.code << endl;
   return 0;
}
```

# Classes and Objects
## Classes as Arguments and Return Values

- Pass address of class object

```
// PTROBJ: pointer parameter & return value
#include <iostream.h>
#include <string.h>
#include “article.h”
article *plargeobj(article *px) // functional spec
{  article *p;                  // pointer
   p = new article;             // new article
   p->codex++;                  // increment code
   strcpy(p->name, px->name);   // copy name
   p->weight = 2 * px->weight;  // double weight
   p->length = 2 * px->length;  // double length
   return p;                    // return new obj.
}
// main to follow
```

# Classes and Objects
## Classes as Arguments and Return Values

- Pass address of class object

```
int main()
{  article s = (246, “Pen”, 20.6, 147.0);
   article *pt;
   pt = plargeobj(&s);
   cout << pt->code << endl;
   return 0;
}
```

# Classes and Objects
## Classes as Arguments and Return Values

- Pass address of class object; V.2

```
// PTROBJ2: pointer parameter
#include <iostream.h>
#include <string.h>
#include “article.h”
void penlargeobj(article *p)    // functional spec
{  p->codex++;                  // modify values
   p->weight *= 2;              // of passed
   p->length *= 2;              // object
}
// main to follow
...
penlargeobj(&s);
```

# Classes and Objects
## Classes as Arguments and Return Values

- Pass address of class object - common error

```
// PTROBJ: pointer parameter & return value
#include <iostream.h>
#include <string.h>
#include “article.h”
article *plargeobj(article *px) // functional spec
{  article obj;                 // local object
   obj.codex = px->code +1;     // increment code
   strcpy(obj.name, px->name);  // copy name
   obj.weight = 2 * px->weight; // double weight
   obj.length = 2 * px->length; // double length
   return &obj;                 // ERROR; WHY?
}
```

# Classes and Objects
## Classes as Arguments and Return Values

- Pass reference to class object

```
// REFOBJ: reference parameter & return value
#include <iostream.h>
#include <string.h>
#include “article.h”
article &rlargeobj(article &x) // functional spec
{  article *p;                  // pointer
   p = new article;             // new article
   p->codex = x.code + 1;       // increment code
   strcpy(p->name, x.name);     // copy name
   p->weight = 2 * x.weight;    // double weight
   p->length = 2 * x.length;    // double length
   return *p;                   // return new obj.
}
// main to follow
```

# Classes and Objects
## Classes as Arguments and Return Values

- Pass reference to class object

```
int main()
{  article s = (246, “Pen”, 20.6, 147.0);
   article *pt;
   pt = &rlargeobj(s);
   cout << pt->code << endl;
   return 0;
}
```

# Classes and Objects
## Classes as Arguments and Return Values

- Dynamic data structures
  - Class member can have any type
  - A member could be a pointer p pointing to another object of the same type (as the one of which p is a member)

```
struct element {int num; element *p;};
```

  - Such types, together with dynamic memory allocation, allow the creation of objects dynamically and the creation of dynamic data structures (e.g. linked lists and binary trees)

# Classes and Objects
## Unions

- Union is a special case of a class
  - so far, all members of class objects exist simulataneously
  - however, if we know that certain members are mutually exclusive we can save space (knowing they can never occur at the same time)
  - Unions allow class objects to share memory space
  - but it is the responsibility of the programmer to keep track of which members have been used.

  - Typically, we do this with a tag field

# Classes and Objects
## Unions

- Union with a tag field

```
enum choice{intflag, floatflag);
struct either_or {
   choice flag;
   union {
      int i;
      float x;
   } num;
}
...
either_or a[100];
a[k].num.i = 0;
a[k].flag = intflag;  // etc.
```

# Classes and Objects
## Member Functions and Encapsulation

- Member functions and encapsulation are features of C++
- With data abstraction (and abstract data types) we identify
  - the set of values a variable of a particular type can assume
  - the set of functions which can operate on variables of a particular type

# Classes and Objects
## Member Functions and Encapsulation

- C++ allows us to localise these definitions in one logical entity: the class
  - by allowing functions to be members of classes (i.e. through encapsulation)
  - by appropriate data hiding

# Classes and Objects
## Member Functions and Encapsulation

```
// VEC1: A class in which two functions are defined
// (inside the class, therefore they act as inline fns)
#include <iostream.h>

class vector {
public:
   float x, y;
   void setvec (float xx, float yy)
   {  x = xx;
      y = yy;
   }
   void printvec() const  // does not alter members
   {  cout << x << ' ' << y << endl;
   }
};
```

# Classes and Objects
## Member Functions and Encapsulation

```
int main()
{  vector u, v;
   u.setvec(1.0, 2.0);  // note form of function call
   u.printvec();
   v.setvec(3.0, 4.0);
   v.printvec();
   return 0;
}
```

# Classes and Objects
## Member Functions and Encapsulation

```
// VEC2: A class in which two functions are declared
// (but defined outside the class)
#include <iostream.h>

class vector {
public:
   float x, y;
   void setvec (float xx, float yy);
   void printvec() const;
};
```

# Classes and Objects
## Member Functions and Encapsulation

```
int main()
{  vector u, v;
   u.setvec(1.0, 2.0);  // note form of function call
   u.printvec();
   v.setvec(3.0, 4.0);
   v.printvec();
   return 0;
}
void vector::setvec (float xx, float yy) // note ::
   {  x = xx;
      y = yy;
   }
void vector::printvec() const
{  cout << x << ‘ ‘ << y << endl;
}
```

# Classes and Objects
## Member Functions and Encapsulation

- Note the use of `vector::`
  - necessary to indicate that the functions are members of the class `vector`
  - as a consequence, we can use the member identifiers (i.e. `x` and `y`)
  - could also have used `this->x` and `this->y` to signify more clearly that `x` and `y` are members of class objects.
  - `this` is a C++ keyword
  - It is always available as a pointer to the object specified in the call to that function

# Classes and Objects
## Member Access Control

- In both previous examples, the scope of the members x and y was global to the function in which vector was declared, i.e. main()
  - x and y could have been accessed by main()
  - this situation may not always be desired
- We would like to distinguish between class members belonging to:
  - the interface ... those that are public
  - the implementation ... those that are private (accessible only to the encapsulated functions)

# Classes and Objects
## Member Access Control

- Private class members are introduced by the keyword `private`
- Public class members are introduced by the keyword `public`
- The default for `structs` (i.e. no keyword provided) is `public`
- The default for `class`es is `private`

# Classes and Objects

## Member Access Control

```
// VEC3: A class with private members x and y

#include <iostream.h>

class vector {
public:
   void setvec (float xx, float yy);
   void printvec() const;
private:
   float x, y;
};
```

# Classes and Objects

## Member Access Control

```
int main()
{  vector u, v;
   u.setvec(1.0, 2.0);  // note form of function call
   u.printvec();
   v.setvec(3.0, 4.0);
   v.printvec();
   return 0;
}
void vector::setvec (float xx, float yy) // note ::
   {  x = xx;
      y = yy;
   }
void vector::printvec() const
{  cout << x << ‘ ‘ << y << endl;
}
```

# Classes and Objects

## Member Access Control

```
// VEC3: A class with private members x and y
// Alternative (but not recommended) declaration
#include <iostream.h>

class vector {
   float x, y; // defaults to private
public:
   void setvec (float xx, float yy);
   void printvec() const;
};
```

# Classes and Objects
## Member Access Control

- Using `::` for functions that return pointers
  - If we are defining a class member function outside the class
    - » e.g. `void vector::setvec()`
  - And if that function returns a pointer...
  - Then the expected `*` goes before the class name
    - » e.g. `char *vector::newvec`
  - and NOT (as might have been anticipated) after the scope resolution operator :: and before the function name
    - » e.g. `char vector::*newvec`

# Classes and Objects
## Constructors and Destructors

- Often, we wish an action to be performed every time a class object is created
- C++ provides an elegant way to do this:
  - Constructor ... action to be taken on creation
  - Destructor ... action to be taken on deletion
- Constructor
  - Class member function with *same name as class*
  - implicitly called whenever a class object is created (defined)

# Classes and Objects
## Constructors and Destructors

- Constructor
  - Class member function with *same name as class*
  - implicitly called whenever a class object is created (defined)
  - no type associated with the function (void, int, etc.)
  - must not contain return statement

# Classes and Objects
## Constructors and Destructors

- Destructor
  - Class member function with *same name as class preceded by a tilde ~*
  - implicitly called whenever a class object is deleted (e.g. on returning from a function where the class object is an automatic variable)
  - no type associated with the function (void, int, etc.)
  - must not contain return statement

# Classes and Objects

## Constructors and Destructors

```
// CONSTR: Demo of a constructor and destructor

#include <iostream.h>
class row {
public:
   row(int n=3)   // constructor with default param = 3
   {  len = n;  ptr = new int[n];
      for (int i=0; i<n; i++)
         ptr[i] = 10 * i;
   }
   ~row()         // destructor
   {  delete ptr;
   }
   void printrow(char *str) const;
private:
   int *ptr, len;
};
```

# Classes and Objects

## Constructors and Destructors

```
void row::printrow(char *str) const
{  cout << str;
   for (int i=0; i<len; i++)
      cout << ptr[i] << ‘ ‘;
   cout << endl;
}
void tworows();
{  row r, s(5);  // two instantiations of row,
                 // one which used default param 3
                 // one which uses 5 as the parameter
                 // Note: can’t write r();
   r.printrow(“r: “);
   s.printrow(“s: “);
}  // destructor ~row() implicitly called on exit

int main()
{  tworows();
   return 0;
}
```

# Classes and Objects
## Constructors and Destructors

- Default Constructor
  - Instead of providing the row() constructor with default argument:
    - » define one constructor with parameters
    - » define another constructor with no parameters, i.e. the default constructor
- If the constructor takes a parameter, then we must provide either a default constructor or a default argument

# Classes and Objects

## Constructors and Destructors

```
// CONSTR: Demo of a constructor and destructor

#include <iostream.h>
class row {
public:
   row(int n)   // constructor with parameters
   {  len = n;  ptr = new int[n];
      for (int i=0; i<n; i++)
         ptr[i] = 10 * i;
   }
   row()   // default constructor
   {  len = 3;  ptr = new int[3];
      for (int i=0; i<3; i++)
         ptr[i] = 10 * i;
   }
   ~row()          // destructor
   {  delete ptr;
   }
   ........
```

# Classes and Objects
## Constructors and Destructors

- Constructor Initializer
  - The `row()` constructor initializes the (private) class object members `len` and `ptr`
  - We can also do this another way using a constructor initializer

```
row(int n=3):len(n), ptr(new int[n])
{  for (int i=0; i<n; i++)
       ptr[i] = 10 * i;
}
```

# Classes and Objects
## Constructors and Destructors

- Constructors and dynamically-created objects
  - when defining class objects
    ```
    row r, s(5);
    ```
  - the constructor `row()` is invoked in the creation
  - we can also create pointers to class objects
    ```
    row *p;
    ```
  - but since this is only a pointer to `row`, the constructor is not called
  - However, if we create the `row` pointed to by `p`
    ```
    p = new row;
    ```
  - the constructor is then called.

# Classes and Objects
## Constructors and Destructors

- Note that the constructor `row()` is not invoked if we use `malloc();`
- Note also that the destructor `~row()` is called when we delete the row object

```
delete p;
```

- but it is NOT invoked if we use `free();`
- We can also specify arguments in the creation:

```
p = new row(5); // 5 element row
```

# Classes and Objects
## Constructors and Destructors

- Constructors and arrays of class objects
  - If we define an array of class objects, the constructor is called for every array element (i.e. for every class object)

```
int main()
{  row a[2], b[6]={5, 1, 2}; //how many rows?
   cout << “Array a (two elements) \n”;
   for (int i=0; i<2; i++)  {
      cout << i;
      a[i].printrow(“: “);
   }
   cout <<“\nArray b (six elements)\n”;
  for (int j=0; j<6; j++) {
      cout << j;
      b[j].printrow(“: “);
   }
}
```

What is the output?

# Classes and Objects
## Constructors and Destructors

– Output:

```
Array a (two elements)
0: 0 10 20
1: 0 10 20

Array b (six elements)
1: 0 10 20 30 40
2: 0
3: 0 10
4: 0 10 20
5: 0 10 20
5: 0 10 20
```

# Classes and Objects
## Operator Overloading and Friend Functions

- We have already seen that we can overload functions
  - must **not** have same number and type of parameters
- We can also overload operators

```
new delete
+   -   *   /   %   ^   &   |   ~
!   =   <   >   +=  -=  *=  /=  %=
^=  &=  |=  <<  >>  >>= <<= ==  !=
<=  >=  &&  ||  ++  --  ,   ->* ->
()  []
```

  - Note that the precedence of the operator cannot be changed

# Classes and Objects
## Operator Overloading and Friend Functions

- Example: vector addition
  - let **u** = ($x_u$, $y_u$) and **v** = ($x_v$, $y_v$)
  - the vector sum **s** = ($x_s$, $y_s$)= **u** + v  is given by

    $x_s = x_u + x_v$

    $y_s = y_u + y_v$

- We will overload the addtion operator + for vectors so that we can write `s = u + v;`

# Classes and Objects

## Operator Overloading and Friend Functions

```
// OPERATOR: an operator function for vector addition

#include <iostream.h>
class vector {
public:
   vector(float xx=0, float yy=0)
   {  x = xx; y = yy;
   }
   void printvec()const;
   void getvec(float &xx, float &yy)const
   {  xx = x; yy = y;
   }
private:
   float x, y;
};
```

# Classes and Objects

## Operator Overloading and Friend Functions

```
void vector::printvec()cost
{  cout << x << ‘ ‘ << y << endl;
}

vector operator+(vector &a, vector &b) //why ref params
{  float xa, ya, xb, yb;
   a.getvec(xa, ya);
   b.getvec(xb, yb);
   return vector(xa + xb, ya + yb); // can’t write a.x
}                                   // and a.y ... why?

int main()
{  vector u(3, 1), v(1,2), s;
   s = u + v;    // sum of two vectors
   s.printvec(); // what’s the output?
   return 0;
}
```

# Classes and Objects
## Operator Overloading and Friend Functions

- Friend Functions
  - recall we couldn't write `a.x` and `a.y` in `operator+` because the members `x` and `y` are private to the class object (and `operator+` is not a class member)
  - consequently we had to have the class member function `getvec()`
  - we can allow `operator+` (and other functions) access to the private members
    - » by defining it as a `friend` function (next)
    - » by having it as a class member function (second next)

# Classes and Objects

## Operator Overloading and Friend Functions

```
// FRIEND: the ‘friend’ keyword applied to an operator
//         function

#include <iostream.h>
class vector {
public:
   vector(float xx=0, float yy=0)
   {  x = xx; y = yy;
   }
   void printvec()const;
   friend vector operator+(vector &a, vector &b);
private:
   float x, y;
};
```

# Classes and Objects

## Operator Overloading and Friend Functions

```
void vector::printvec()cost
{  cout << x << ‘ ‘ << y << endl;
}

vector operator+(vector &a, vector &b)
{
   return vector(a.x + b.x, a.y + b.y); //friend access
}

int main()
{  vector u(3, 1), v(1,2), s;
   s = u + v;    // sum of two vectors
   s.printvec(); // what’s the output?
   return 0;
}
// NOTE: operator+ is a friend function but NOT a class
  member
```

# Classes and Objects

## Operator Overloading and Friend Functions

```
// FRIEND: the 'friend' keyword applied to an operator
//         function
//         This time, define operator+ in the class

#include <iostream.h>
class vector {
public:
   vector(float xx=0, float yy=0)
   {  x = xx; y = yy;
   }
   void printvec()const;
   friend vector operator+(vector &a, vector &b)
   {
      return vector(a.x + b.x, a.y + b.y);
   };
private:
   float x, y;
};
```

# Classes and Objects
## Operator Overloading and Friend Functions

- Operators as member functions
  - we can also allow `operator+` access to the private members
    - » by defining it as a class member
    - » however, the syntax is a little odd!
    - » operator+ is a binary operator but it is allowed have only one parameter (the second operand)
      - the first operand is accessed implicitly and directly

# Classes and Objects

## Operator Overloading and Friend Functions

```
// OPMEMBER: An operator function as a class member

#include <iostream.h>

class vector {
public:
   vector(float xx=0, float yy=0)
   {  x = xx; y = yy;
   }
   void printvec()const;
   vector operator+(vector &b);
private:
   float x, y;
};
```

# Classes and Objects

## Operator Overloading and Friend Functions

```
void vector::printvec()cost
{  cout << x << ‘ ‘ << y << endl;
}

vector vector::operator+(vector &b)
{
   return vector(x + b.x, y + b.y);//first operand is
}                                  //the vector for
                                   //which the function
int main()                         //is called
{  vector u(3, 1), v(1,2), s;
   s = u + v;    // sum of two vectors
   s.printvec(); // what’s the output?
   return 0;
}
```

# Classes and Objects
## Operator Overloading and Friend Functions

- in effect

```
s = u + v;
```

is equivalent to

```
s = u.operator+(v);
```

which is why there is only one operand for a binary operator!

# Classes and Objects
## Operator Overloading and Friend Functions

- Note that we are not always free to choose between a member function and a friend function for operator overloading:
- C++ requires that the following operators can only be overloaded using member functions (we cannot define friend functions for them)

```
=,  [],  (),  ->
```

# Classes and Objects
## Operator Overloading and Friend Functions

- Overloading applied to unary operators
  - Define the minus sign as the unary operator for vectors:

```
vector u, v;
...
v = -u;
```

  - and, from which, we can then proceed to define a binary minus operator since:

```
a - b = a + (-b)
```

# Classes and Objects

## Operator Overloading and Friend Functions

```
// UNARY: An unary operator, along with two binary ones

#include <iostream.h>

class vector {
public:
   vector(float xx=0, float yy=0)
   {  x = xx; y = yy;
   }
   void printvec()const;
   vector operator+(vector &b); // binary plus
   vector operator-();          // unary minus
   vector operator-(vector &b); // binary minus

private:
   float x, y;
};
```

# Classes and Objects

## Operator Overloading and Friend Functions

```
void vector::printvec()cost
{  cout << x << ‘ ‘ << y << endl;
}

vector vector::operator+(vector &b) // Binary plus
{
   return vector(x + b.x, y + b.y);
}

vector vector::operator-() // Unary minus
{
   return vector(-x, -y);
}

vector vector::operator-(vector &b) // Binary minus
{
   return *this + -b; // recall ‘this’ is a pointer to
}                     // the current object
```

# Classes and Objects

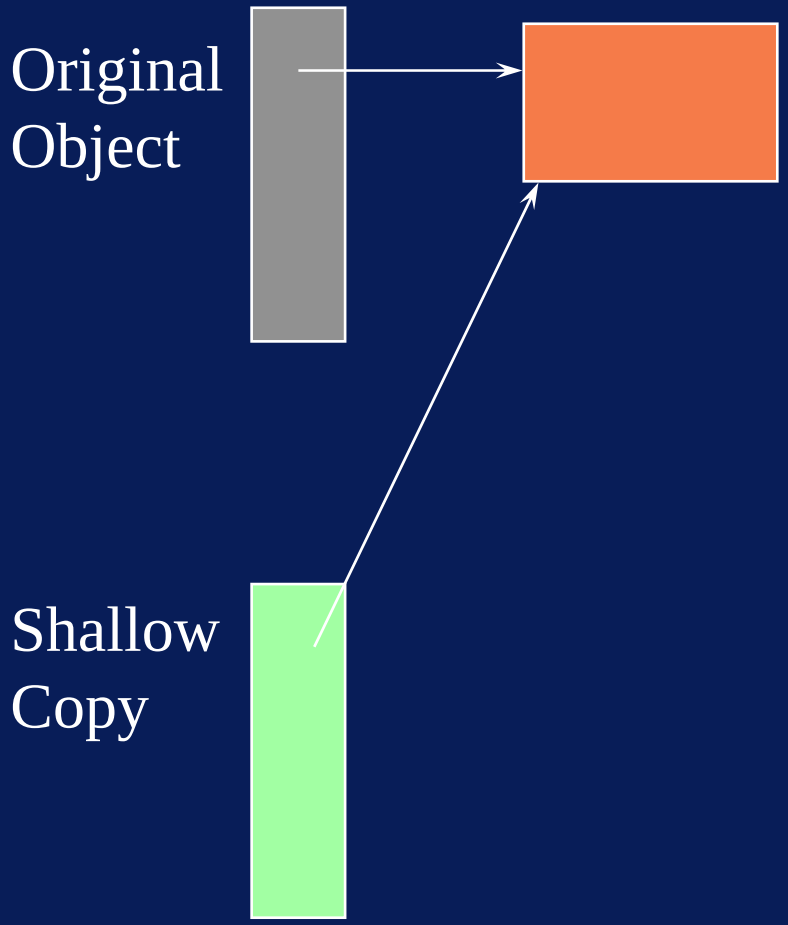
## Operator Overloading and Friend Functions

```
int main()
{  vector u(3, 1), v(1,2), sum, neg, diff;
   sum = u + v;
   sum.printvec();  // what's the output?
   neg = -sum;
   neg.printvec();  // what's the output?
   diff = u - v;
   diff.printvec(); // what's the output?
   return 0;
}
```

# Classes and Objects
## Copying a Class Object

- A class object that contains a pointer to dynamically allocated memory can be copied in two ways:
  - Shallow copy
    - » where the class contains only member functions and ‘simple’ data members (which are not classes)
    - » copying is by default done ‘bitwise’
      - all members, including the pointers, are copied literally

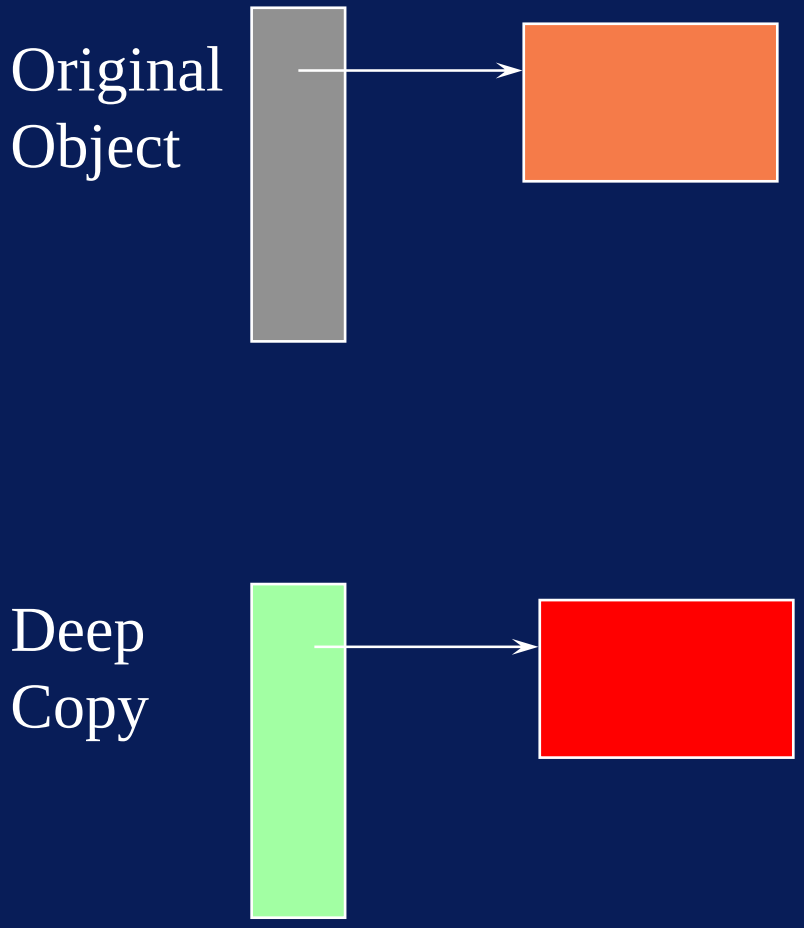
# Classes and Objects
## Copying a Class Object



# Classes and Objects
## Copying a Class Object

- A class object that contains a pointer to dynamically allocated memory can be copied in two ways:
  - Deep copy
    - » all members are copied
    - » but the pointer and the data to which it points are replicated

# Classes and Objects
## Copying a Class Object



# Classes and Objects
## Copying a Class Object

- The difference between deep and shallow copies is important when the referenced memory is allocated by a constructor and deleted by a destructor
  - why?
  - because the shallow copies will also be effectively deleted by the destructor (and, anyway, attempting to delete the same thing twice is dangerous and illegal)

# Classes and Objects
## Copying a Class Object

- If we require deep copies, then we must take care to define the constructors and assignments appropriately
- Typically, we will define a copy constructor as a (function) member of a class for copying operations other than by assignment
  - This copy constructor will be declared in the following way:

```
class_name(const class_name &class_object);
```

  - parameter is always a reference parameter

# Classes and Objects
## Copying a Class Object

- For copying by assignment, we must define an assignment operator to prevent shallow copying

```
class_name operator=(class_name class_object);
```