Using Classes in C++

Note: Many of the examples given below are taken from the Irvine book

I. Creating Classes
   A. An Example [from Deitel & Deitel]: structs vs. classes
      Drawbacks to using a struct:
      1. Initialization not specifically required, and may be done incorrectly
      2. Invalid values can be assigned to struct members (as in example)
      3. If struct implementation changes, all the programs using the struct must also change
      4. Structs aren’t first-class: they cannot be printed as a unit or compared

   B. Class Definition
      Basic form is:
      ```
      class class-name {
      member-list
      }
      ```
      Class name has file scope and external linkage. E.g. Point class for points on a 2D cartesian graph:
      ```
      class Point {
      public:
      int GetX(); // Returns the value of x.
      void SetX(); // Sets the value of x.
      private:
      int x;    // x-coordinate
      int y;    // y-coordinate
      }
      ```
      Keywords public and private are access specifiers. Note that storage is not allocated until an instance of the class is created (there is also an access specifier protected, which we will avoid for now)

   C. Class Objects
      General format is: class-name identifier. We can access class methods using the dot operator ("."), E.g.
      ```
      Point P;   // Note this is different than Point P();
      P.SetX( 300 );
      cout << "x-coordinate is: " << P.GetX();
      ```
      Copying from one class object to another by default does a bit-wise copy, such as
      ```
      Point P1, P2;
      ...
      P2 = P1;
      ```

   D. Controlling Member Access
      A member may be either a function or a data item
      1. Access specifiers:
         a. public: non-member function can access these members
         b. private: member of same class only can access these members
         c. protected: same as private, only derived classes can access these members
      2. If no access specifier is given, the default is private.
      3. Access specifiers may be listed multiple times, though good style dictates that all public members are given first, then private.
E. Example: the Point class

```c++
class Point {
public:
    intGetX() // Returns value of X
    {
        return x;
    }

    void SetX(int xval) // Sets value of X
    {
        x = xval;
    }
private:
    int x; // x-coordinate
    int y; // y-coordinate. Note y funcs. not shown
}; // End of Point class
```

Short functions as shown above may be given all on one line.

F. Types of Member functions

1. Accessor, or selector functions (e.g. GetX)
2. Mutator functions (e.g. SetX)
3. Operators that let you define C++ operators for each class object (e.g. move a point)
4. Iterators that process a collection of objects (e.g. recompute each Account balance)

G. Inline and Out-of-line definitions

Rather than list a member function definition along with the declaration under the public access specifier, we can leave the member function declaration as public but hide the implementation details. E.g.

```c++
class Point {
public:
    void SetX(int xval); // Sets value of X.
...
private:
    int x; // x-coordinate
    ...
}; // End of Point class
```

1. The binary scope resolution operator "::" specifies that Setx corresponds to the Point class. It can access the private data members only because it has this correspondance.
2. The member function SetX must specify the type of parameters, though parameter names need not be given. (In fact, given parameter names need not be the same as in the definition) Since this declaration specifies the interface, it is a good idea to provide the parameter names as well as brief documentation.
H. Constructors

A constructor is a special-purpose member function that is automatically executed when a class object is created. [Irvine] The constructor always has the same name as its class, with no return type.

1. A constructor with no parameters is called a default constructor. E.g.

```cpp
class Point {
public:
    Point() // Default constructor
    {
        x = 0;
        y = 0;
    }
private:
    int x;
    int y;
}; //End of Point class
```

So now when executing your program, the declaration `Point aPoint;` has the effect of creating a point instance called `aPoint` with its x & y data members initialized to 0.

2. Alternate & Overloaded constructors

We can have more than one constructor, where we also provide parameters to initialize an object's data members, optionally giving default values. E.g.

```cpp
Point::Point( int xval=0; int yval=0)
{
    SetX( xval);
    SetY( yval);
}
```

where in the main part of your program you could then have:

```cpp
Point A;  // initialized to 0,0
Point B(3); // initialized to 3,0
Point C(3,5); // initialized to 3,5
```

a. Rather than `SetX(xval)` we could simply have `x=xval`, since member functions have access to private data members. Using the Accessor & Mutator functions is better since:

   (1). It allows us to later change the private data (e.g. rename x to `pointx`) without worrying about any side-effects within the class implementation.

   (2). It allows us to later implement validity checks on member data values

b. See examples of RogueWave date constructors
3. Copy constructors

A *copy constructor* is automatically called anytime an object is passed by value. It makes a local copy of the object, making a bitwise copy by default (the system makes a default constructor).

a. Both declarations of Q and Z below result in a call to the copy constructor:
   
   ```
   Point P;
   Point Q(P);
   Point Z = P;
   ```

   where the copy constructor could look like:
   ```
   Point::Point( const Point & p2)
   {
       x = p2.x;
       y = p2.y;
   }
   ```

   The reference parameter is *const* qualified so that the original is unchanged.

I. Array of Class Objects

   The constructor for each member is called upon definition of the array, E.g.
   ```
   Point figure[3];
   ```

   calls the Point constructor for each of the 3 members.

J. Destructors

   1. This special function is automatically called when a class object is deleted. This can happen in these situations:
      
      a. An object has local scope and goes out of scope
      b. An object has file scope and the main program ends
      c. An object was dynamically allocated (using *new*) and is now deleted (using *delete*)

   2. The name of a destructor is the class name preceded by a tilde (~). E.g.
   ```
   class Point {
   public:
       ~Point() { cout<< "Point destructor called\n"; }
       ...
   }; // End of class Point
   ```
II. Miscellaneous

A. The ability to have objects be "1st-class" in C++ is not automatic. The different aspects of an object being first class must be implemented by the user. This includes constructors, destructors, overloading output & input operators, overloading other operators, the copy constructor. Functions must be written for each of these.

B. Class scope

1. Class data members and member functions are immediately accessible by that class’s member functions simply by name
2. If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden.

C. Difference between constructor and a function call

Given the code below:

```cpp
class Date {
public:
    Date();
    Date(int d, int m=0, int y=0);
private:
    int day, month, year;
};

Date::Date() //Default constructor
{
    day = 0, month = 0; year = 0;
}

Date::Date(int d, int m, int y)//Constructor
{
    day = d; month = m, year = y;
}

int main()
{
    Date d1;    // Default constructor
    Date d2(2);
    Date d3(24,2);
    Date d4(24,2,97);
    Date d5();  // Function declaration
    ...
}

Date d5() //Function definition
{
    return Date( 24, 2, 97);
}
```

Note that compiler must check arguments to tell the difference between

Date d4( 24, 2, 97)

and the function declaration

Date d4( int, int, int);

where the typenames tell the compiler that this is a prototype
D. Preventing multiple inclusions of a header file
   ifndef TIME1_H
   define TIME1_H
   ...
   endif

E. delete - make sure you use [] with delete if it is a dynamic array. This way it calls the destruc-
   tor for you properly. E.g.
   int *arrayPtr;
   arrayPtr = new int[ 100]; // Dynamic array
   ...
   delete [] arrayPtr;

F. References and Pointers
   1. A reference parameter means the parameter may be changed
   2. A reference variable essentially is the object it is referencing
   E.g.
   // FIG5_4.CPP, Deitel & Deitel
   // Demonstrating the class member access operators . and ->
   #include <iostream.h>

   class Count { // Simple class Count
      public:
         int x;
         void print() { cout << x << '
'; }
   };

   main()
   {
      Count counter,                // create counter object
         *counterPtr = &counter, // pointer to counter
         &counterRef = counter;  // reference to counter

      cout << "Assign 7 to x and print using the object's name: ";
      counter.x = 7;       // assign 7 to data member x
      counter.print();     // call member function print

      cout << "Assign 8 to x and print using a reference: ";
      counterRef.x = 8;    // assign 8 to data member x
      counterRef.print();  // call member function print

      cout << "Assign 10 to x and print using a pointer: ";
      counterPtr->x = 10;  // assign 10 to data member x
      counterPtr->print(); // call member function print

      return 0;
   }

   /* --------- Output ------------
   Assign 7 to x and print using the object's name: 7
   Assign 8 to x and print using a reference: 8
   Assign 10 to x and print using a pointer: 10
   */
Constructors are called when object enters scope. Destructor called when it exits scope.

// FIG16_9.CPP, Deitel & Deitel, Constructors & destructors order
#include <iostream.h>

class CreateAndDestroy {
    public:
        CreateAndDestroy(int);  // constructor
        ~CreateAndDestroy();    // destructor
    private:
        int data;
    };

CreateAndDestroy::CreateAndDestroy(int value)
    { data = value;
        cout << "Object " << data << " constructor"; }

CreateAndDestroy::~CreateAndDestroy()
    { cout << "Object " << data << " destructor " << endl; }

void create(void) // Function to create objects
    {
        CreateAndDestroy fifth(5);
        cout << " (local automatic in create)\n";
        static CreateAndDestroy sixth(6);
        cout << " (local static in create)\n";
        CreateAndDestroy seventh(7);
        cout << " (local automatic in create)\n";
    }

CreateAndDestroy first(1);        // global object

main()
{
    cout << " (global created before main)\n";
    CreateAndDestroy second(2);        // local object
    cout << " (local automatic in main)\n";
    static CreateAndDestroy third(3); // local object
    cout << " (local static in main)\n";
    create();  // call function to create objects
    CreateAndDestroy fourth(4);    // local object
    cout << " (local automatic in main)\n";
    return 0;
} /* Output starts on line below
Object 1 constructor (global created before main)
Object 2 constructor (local automatic in main)
Object 3 constructor (local static in main)
Object 4 constructor (local automatic in create)
Object 5 constructor (local automatic in create)
Object 6 constructor (local static in create)
Object 7 constructor (local automatic in create)
Object 7 destructor
Object 5 destructor
Object 4 destructor
Object 2 destructor
Object 3 destructor
Object 1 destructor
Object 6 destructor */

Static objects exist until program terminates.
H. Returning a reference to a private data member.

1. It may be desirable to return a reference, such as when we want to use the return value on the left-hand side (l-value) of an assignment or when we want to chain assignments together such as in \( A=B=C \);

2. This could unfortunately give public access to a private variable as follows:

```cpp
// Fig. 16.11, Deitel & Deitel, using the Time class

class Time {
public:
    Time(int = 0, int = 0, int = 0);
    void setTime(int, int, int);
    int getHour();
    int &badSetHour(int); // DANGEROUS reference return
private:
    int hour;
    int minute;
    int second;
};

// BAD PROGRAMMING PRACTICE: Returning a ref. to private data
int &Time::badSetHour(int hh)
{
    hour = (hh >= 0 && hh < 24) ? hh : 0;
    return hour; // DANGEROUS reference return
}

int main()
{
    Time t;
    int &hourRef = t.badSetHour(20);

    cout << "Hour before modification: " << hourRef << \n;
    hourRef = 30; // modification with invalid value
    cout << "Hour after modification: " << hourRef << \n;

    // Dangerous: Returns a reference can be used as an lvalue.
    t.badSetHour(12) = 74;
    cout << \nBAD PROGRAMMING PRACTICE!!!!!!!!!!\n" << "badSetHour as an lvalue, Hour: " << t.getHour() << \n"
    return 0;
}

/* Output:
Hour before modification: 20
Hour after modification: 30
BAD PROGRAMMING PRACTICE!!!!!!!!!!
badSetHour as an lvalue, Hour: 74
*/
```
I. Constant objects and const member functions

1. *const* could be used in various places in a member function.

   E.g. a function to give a date that has only the common elements between two other dates

   ```cpp
   const Date & HashTable::findCommonElements(const Date & theDate) const
   ```

   Note it might be desirable to return a constant reference to use as an *r-value*

2. C++ compilers respect *const* declarations so rigidly that they disallow member function calls for *const* objects, unless the member function itself is *const*. E.g.

   ```cpp
   // Modified from Fig. 17.11, Deitel & Deitel, using const
class Time {
  public:
    Time( int h = 0, int m = 0, int s = 0); //Default constructor
    void setHour( int); // set hour
    ...
    int getHour() const; // get hour (note *const*)
  private:
    int hour, minute, second;
};

int Time::getHour() const { return hour; }// must have *const*)
```

3. A *const* object can not be modified by assignment, so it must be initialized, such as we did in the example above. If any of our private data members themselves are *const*, we must use what’s called a *member initializer list* and not an assignment to do this. E.g.

   ```cpp
   class Date {
  public:
    Date(int d=21, int m=9, int y=63);// constructor
    ...
  private:
    int day;
    const int month, year;// month and year are constant
  };

  Date::Date(int d, int m, int y) // constructor
    : month( m), year( y) // member initializer list
  {
    day = 0;
  }

  int main()
  {
    Date d1( 15, 12, 1997); // constructor called
    ...
  }
  ```

   The syntax "month( m)" means initialize data member "month" to have the value "m". If the constructor simply used assignments in the body (day = month = year = 0;) we would get a compiler error: *assignment of read-only member*
4. A \textit{const} member function can be overloaded with a non-\textit{const} version. The choice of which one to use is done by the compiler depending on whether or not the caller is \textit{const}.

J. Composition: Using classes in other classes
Example: Creating an Employee class that uses a Date class for start date & birth date
[See Overhead]
1. Note that Employee initializers invoke Date constructors, using the Employee parameters.
2. Member objects are constructed before their enclosing class objects are.

K. Initializer lists -vs.- body assignment in constructors
1. Remember, a default constructor is necessary in situations where you will create an array of members of your class, such as: Date birthdays[5];
2. In cases where a member is itself an instance of a class (e.g. \texttt{Employee} includes field \texttt{startdate} of type \texttt{Date}), if that member is not initialized using the initializer list, then the member object’s default constructor will be called automatically. E.g.

```cpp
class Employee {
public:
    Employee(char *, char *, int, int, int, int, int, int); // hireDate(hmonth, hday, hyear) Default const. called!
    void print() const;
private:
    char lastName[25];
    char firstName[25];
    Date birthDate;
    Date hireDate;
};

Employee::Employee(char *fname, char *lname, int bmonth, int bday, int byear,
    int hmonth, int hday, int hyear) : birthDate(bmonth, bday, byear) // hireDate(hmonth, hday, hyear) Default const. called!
{
    hireDate.setDate( hmonth, hday, hyear);
    ... // hireDate(hmonth, hday, hyear). This date is then assigned the value it is supposed to have using setDate.
}
```

a. By the time the Employee constructor body is entered, the day, month, and year fields of \texttt{hireDate} have already been initialized by the default constructor. This date is then assigned the value it is supposed to have using \texttt{setDate}.

L. Creating an array of objects where the objects have no default constructor can cause problems [see overhead using RogueWave Dates]
M. Friend functions and classes
It is possible to give non-class functions the access rights of class member functions by declaring them as friends. E.g.

```cpp
// FIG17_5.CPP, Deitel & Deitel
// Friends can access private members of a class.
#include <iostream.h>

class Count {
    friend void setX(Count &, int);  // friend declaration
public:
    Count() { x = 0; }               // constructor
    void print() const { cout << x << endl; }  // output
private:
    int x; // data member
};

// Can modify private data of Count because it is a friend
void setX(Count &c, int val)
{
    c.x = val; // legal: setX is a friend of Count
}

main()
{
    Count object;

    cout << "After instantiation, object.x is: ";
    object.print();
    setX(object, 8);    // set x with a friend
    cout << "After friend function call, object.x is: ";
    object.print();

    return 0;
}
```

a. Although friendship declarations can be placed anywhere, it is a good practice to place them before the first access specifier (e.g. public)
b. To declare an entire class as a friend, use a declaration such as:
   friend theFriendClass;
   as a member of the class doing the declaration.
c. If function "setX" above were not declared as "friend," then the compiler would give warnings and errors.
d. Be aware that friendship declarations can weaken the advantages of information hiding in the OOP approach.
N. More comments on using *this

Remember that *this refers to the implicit argument.

1. *this can be used as a return value to allow chaining. E.g. assuming the Time class:
   
   ```
   Time &Time::setHour(int h)
   {
       hour = (h >= 0 && h < 24) ? h : 0;
       return *this; // enables chaining
   }
   ```

   and assuming similar definitions for setMinute() and setSecond(). This would allow us to type:

   ```
   Date t;
   ...
   t.setHour(18).setMinute(30).setSecond(22);
   ```

   Note that the dot operator associates from left to right, so setHour() is done first and so on. This is important if we had

   ```
   t.setHour(18).printStandard();
   ```

   where printStandard() does not return *this, since reversing the order of the two member functions would give an error.

2. Another use is to make sure we don’t assign an object onto itself, which could cause errors when the objects contain pointers to dynamically allocated storage. E.g.

   ```
   if ( parameterObject != *this)
   // Do the assignment
   ```

O. Templates

We can create an ADT without knowing what type of object will be operated on. We leave the object type as a parameter.

1. E.g. a Stack template [See overhead]

2. Declaring a template looks like a class with the addition of

   ```
   template <class T>
   ```

   which allows us to specify the object type as a parameter T which can then be used throughout the class header and member function definitions

3. The member function definitions similarly each begin with the header

   ```
   template <class T>
   ```

4. The driver (main) then creates an instance of a floating point stack & initializes its values, then creates an instance of an integer stack and uses it

5. For instance, declaring

   ```
   stackPtr = new T[size];
   ```

   when T is int becomes

   ```
   stackPtr = new int[size];
   ```
P. Static Class Members

1. Normally each class instance has its own copy of class data members. The exception is a static data member, where there is only one of these for the entire class. E.g.

   // Simple class Array (for integers), Deitel & Deitel
   class Array {
   public:
      Array(int arraySize = 10); // default constructor
      ~Array();                // destructor
      ...
   private:
      int *ptr; // pointer to first element of array
      int size; // size of the array
      static int arrayCount; // # of Arrays instantiated
   };

2. Static data members must be initialized at file scope, E.g.

   int Array::arrayCount = 0;

3. Static data members exist even when no objects of the class exist, however in this situation they can only be accessed:
   a. by prefixing the class name and binary scope resolution operator to the name if the static member is public, E.g. Array::arrayCount
   b. by using a static member function if the data member is private, E.g.
      int Array::getArrayCount() { return arrayCount; }

4. A Static data member can be accessed directly by member functions, as in

   // Destructor for class Array
   Array::~Array()
   {
      --arrayCount; // one fewer objects
      delete [] ptr; // reclaim space for array
   }

5. A member function accessing only static class members may itself be declared static

6. A static member function has no this pointer since static members exist independent of objects of the class.
III. Operator Overloading

A. Background

1. We’ve seen this in C, using + for both integer & real
2. Allows us to make our own classes "first-class"
3. We can implement the same functionality by either overloading an operator for the class or by an explicit file-scope function call. We overload if it will make the code more clear. Examples:
   a. We could have
   ```cpp
class String {
   public:
       String & String::operator+=( const String &);
   ...
   }
   or
   class String {
   public:
       friend String & String::operator+=( String &, const String &);
   ...
   }
   where then concatenating two String class instances (s1 += s2) would be translated by the compiler into either
   ```cpp
   s1.operator+=(s2);
   or
   ```cpp
   operator+=( s1, s2);
   ```cpp
   depending on which one was defined (both can’t be defined).
B. Overloading an operator is done by making the member function name the keyword `operator` followed by the symbol being overloaded
C. To use a symbol with a class object you must always overload the operator, with the following two exceptions which still have meaning if not overloaded:
D. The assignment operator ":=" (does a member-wise copy)
E. The address operator "&" (returns address of object in memory)
F. Restrictions on overloading
1. It is not possible to create new operators. Only existing ones may be overloaded. (e.g. +, -, *, /, =, ==, ...)
2. Some operators may not be overloaded (., *, ::, ?:, sizeof)
3. Overloading can not change precedence, associativity, or number of operands.
4. Meaning of built-in types can not be changed by overloading (e.g. int + int always will mean the same thing)
5. Overloading assignment ":=" and addition "+" does not imply that "+=" is defined. E.g.
   ```cpp
   object2 = object2 + object1;
   ```cpp
   overloading to implement
does not imply that
   ```cpp
   object2 += object1;
   ```cpp
   is automatically defined
G. Overloaded operator member functions:
1. Must have the left-most operand be a class object of the operator’s class. Otherwise it must be implemented as a non-member function or as a `friend` function.
2. A non-member non-friend function would need to use the class’s accessor functions to have access to private data members
H. Overloading for *cin* and *cout*
1. *cout* has already been overloaded to handle the basic types. E.g.

```cpp
int x=5;
real pi=3.1415;
cout<< x << pi;
```

ends up being two function calls, one to

```
ostream::operator<<( int);
```

and another one to

```
ostream::operator<<( real);
```

Note that these both return (*this) to allow chaining.

2. Example to read in phone number & print it out [Overhead]
   a. Note that overloading input and output here means we must make the functions _friends_ (since left-most operand isn’t a class object of the operator’s class)
   b. input.ignore( n) Has the effect of ignoring n characters from input stream _input_
   c. input.get(string, size) Has the effect of storing size-1 characters from input stream _input_ into _string_
   d. cin >> phone gets translated by the compiler into operator>>(cin, phone)
      or the equivalent cin.operator>>(phone), which shows that the operator overloaded function belongs to the iostream _cin_, not to _phone_. This is why it must be a friend.

I. Overloading operators in the _Array_ class [Overhead]
1. Does range checking of subscripts
2. Overloads stream input (extraction) & output (insertion) for Arrays
3. The overloaded subscript operator returns a reference (see driver _main_ for examples)
   a. This reference can be used to retrieve the value at that subscripted location (used as an r-value)
   b. The reference can also be used as an l-value, which allows it to be used on the left of an assignment.
   c. It also catches invalid subscripts
4. Could implement _operator!=_ as the opposite of _operator==_ as follows:
   ```cpp
   int Array::operator!=(const Array &right) const
   {return !( (*this)==right ); }
   ```
5. Assignment operator
   a. The expression integers1 = integers2 gets compiled into
      ```cpp
      integers1.operator=( integers2)
      ```
   b. An overloaded assignment provides transparent resizing. Note also that it prevents self-assignment. (Trace to show this problem)
6. The _copy constructor_
   a. This is called in situations when a copy is needed, such as:
      1. Call by value (more on this later)
      2. Returning an object from a called function
      3. Initializing an object to be a copy of another object of the same class,
         E.g.
         ```cpp
         Array integers3( integers1);
         Array integers3 = integers1;
         ```
   b. Why copy each member instead of just copying the pointer, such as
      ```cpp
      ptr = init;
      Answer: eventually the destructor for one of them would have the effect of also destroying the other, probably causing a run-time error.
      ```
7. Relationship between the copy constructor and the overloaded assignment operator
   a. Can extract common code into a private member function that both call. E.g.
   
   ```cpp
   private:
       ...
       void copyObject( const Array & right); // used by copy const.
   
   // Code used by both copy const. and overloaded assignment (=)
   void Array::copyObject( const Array & init)
   {
       size = init.size; // size this object
       ptr = new int[size]; // create space for array
       assert(ptr != 0); // terminate if memory not allocated

       for (int i = 0; i < size; i++)
           ptr[i] = init.ptr[i]; // copy init into object
   }
   ``

   // Copy constructor for class Array
   Array::Array(const Array &init)
   {
       ++arrayCount; // count one more object
       copyObject( init); // could also be (*this).copyObject( init)
   }

   // Overloaded assignment operator
   const Array &Array::operator=(const Array &right)
   {
       if (&right != this) // check for self-assignment
       {
           delete [] ptr; // reclaim space
           copyObject( right);
       }

       return *this; // enables x = y = z;
   }

J. Why a copy constructor is necessary

1. Assume we have no copy constructor and we are given a linked-list class called List, declared as: [Horstmann]

```cpp
class List {
private:
    class Node { // Class within a class
        public:
            int info;
            Node* next;
        }
    Node* head;
    int length;
};
```

Note constructor `List::Node::Node( int i, Node* n) : info(i), next(n){}`

2. Now let's say we use this in a function call:

```cpp
double average( List a) {
    double sum = 0;
    int n = a.getLength(); // Assume this is defined
    ... return sum/n;
} // ~List destroys a ...
```

List alist;
double avg = average( alist);

3. The function call has the effect of copying the argument into the parameter, and allocating memory for the function’s local variables on the stack. This could give us the situation:

4. When function `average` exits, all local variables on the stack are destroyed. Memory for the base-typed variables `sum` and `n` are abandoned on the stack, but the `~List` destructor is called for `a` as `a` goes out of scope. This destroys the entire list.

   a. Upon returning to the caller, now `alist` has been deleted as a side-effect of the function.
   b. Eventually when the destructor is called for `alist`, an invalid pointer will be sent to `delete` and the free memory list could be corrupted.

5. Why not have the compiler use the overloaded assignment operator to assign `a = alist` when the function is called to take care of this problem?

   a. The first thing an assignment operator does is to delete the memory for the object on the l.h.s. This would cause a problem here, since the l.h.s. `(a)` is not defined, so calling delete on its `head` pointer could again corrupt the free list.
   b. The assignment operator can only be used to copy into an object that already exists, so we need a `copy constructor` to initialize a new object as a copy of an existing one.
   c. Note the difference between: "List b = a;" and "List b; b = a;"
K. Comments on destructors
1. Pointers have no destructors. E.g.

```cpp
void someFunction()
{
    Employee* pemployee1 = new Employee();
    Employee employee2();
    ...
} // ~Employee destroys employee2, but not pemployee1;
```

Similarly when a reference goes out of scope, no destructor is triggered

```cpp
void print( List a, ostream& os)
{
    ...
} //~List destroys a, but os is not destroyed
```

2. Can explicitly call destructor using `delete`, or it can be invoked by the system when an object goes out of scope.

3. Can only deallocate from heap, not from stack. E.g. [Horstmann]

```cpp
Employee joe("Joe User");
Employee* pj = &joe;
Employee* ph = new Employee("Harry Hacker");
Employee* pk = ph;
delete pk;    // OK, calls destructor ~Employee
delete ph;   // Error, deleting twice
delete pj;   // Error, cannot delete pointer to stack, only to heap
```

4. Destruction appears to be incompatible with assignment, thus showing the need for the overloaded assignment operator. E.g. Assuming we have no overloaded assignment operator, consider the code:

```cpp
List a;
    a.insert(...);
...
List b;
    b.insert(...);
...
    a = b;///< Assignment operator called
    // Assume that then b goes out of scope
```

Pictorally this could look like:

The lists before assignment:

The lists after assignment:

The lists after destruction of b:

a. Neither a nor b are now accessible
b. Subsequently calling ~List on a could corrupt the free list
c. Having an overloaded assignment operator correctly deletes the destination, allocates memory for the new destination, and does a deep copy
L. Type Conversions.
1. Consider the fraction class from [Horstmann]:

```cpp
class Fraction {
public:
    Fraction( long n = 0, long d = 1);
    Fraction operator+( Fraction) const;
    ...
private:
    long num;
    long den;
};

Fraction::Fraction( long n, long d)
: num(n), den(d)
{}

Fraction Fraction::operator+(Fraction b) const
{
    return Fraction( num * b.den + den*b.num, den * b.den);
}
```

a. Note that `ClassName instance = integer` is the same as `ClassName instance( integer)`
b. We can convert an integer into a fraction using the constructor
   ```cpp
   Fraction f = 2;
   ```
c. Similarly this conversion is carried out in the arithmetic expression
   ```cpp
   Fraction g = f + 2;
   // same as g = f.operator+(Fraction( 2,1));
   ```
d. However the expression
   ```cpp
   2 + f
   ```
   is in fact not legal if `operator+` is an operation of the Fraction class, since the compiler tries to do
   ```cpp
   2.operator+(Fraction f)
   ```
   where "2.operator+" is the addition operator for integers. The compiler does not do:
   ```cpp
   Fraction(2).operator+( Fraction f)
   ```
e. The implicit argument is used for class selection, and no conversion on it is ever attempted. (automatic conversion is not symmetric)
f. In an example such as this where we would like conversion to be done on either argument, the operator is best defined as a regular function:
   ```cpp
   Fraction operator+( Fraction a, Fraction b)
   ```
2. Constructors with a single argument can serve as type conversion functions, e.g.
   ```cpp
   Fraction( int)
   ```
could convert from an integer to a fraction with denominator of 1.
3. There are two cases where we must do something different:
   a. The target type is a class that you have no authority to modify
b. The target type is not a class. E.g.

class Fraction {
public:
  ...
  operator double() const; // cast into double operator
};

Fraction::operator double() const
{    return (double) num / (double) den;
}

...
Fraction f(1,4);
double y = sqrt(f); // y = sqrt(f.operator double())

4. Another example for a string class [Deitel & Deitel]
   a. Note the conversion constructor with the default of the empty string:
      String(const char * = ""); // conversion constructor
      (1). Having this means it isn’t necessary to have a separate overloaded
          assignment operator to assign from a conventional string into a String
          object. We can simply have
          String s1;
          s1 = "a conventional string";
      or similarly
          String s2("another conventional string");
      (2). Only a single constructor can be used in this manner to try to match the
          needs of the overloaded assignment operator. It is not possible to
          match an overloaded operator’s needs by performing a series of
          implicit user-defined conversions.
   b. See also the string concatenation operator
      String &operator+=(const String &); // concatenation
      where s1+=s2 would get compiled into s1.operator+=(s2)
   c. How about concatenation of a String and a char *? The conversion constructor
      would take care of this
   d. See one of the comparison operators (e.g. operator ==)
   e. Particularly note the overloaded function call operator
      String &operator[](int, int); // return a substring
      used to implement substring.
      (1). We could have multiple versions of this corresponding to functions
          with various argument signatures
      (2). We could use this to change multiple-dimensioned arrays syntax from
          the usual array[i][j] to be array(i,j)