Chapter 10: Construction and Destruction

The way of existence in C++:

> **Because of my title I was the first to enter here. I shall be the last to go out.**—The Duchess d'Alencon, refusing help during a fire at a charity bazaar in 1897.

10.1 New C++ Concepts

Dumps. Some of the most troublesome program bugs are caused by mishandling storage allocation, or failure to understand what happens, when, and why. During the long process of mastering C++, it is a good idea to track the constant allocation and deallocation actions:

- By printing comments in constructors and destructors.
- By using dump functions, as in the Van and Box classes.
- By printing an object (or part of it) when it is created.

10.1.1 Talking About Yourself

The keyword `this` is used to refer to the implied paramter within a class function. In class T, the type of `this` is `T*` because "`this`" is a self-pointer; it points at the implied argument. Thus, "`cout << this`" prints the address of the current object (in hexadecimal) and "`cout << *this`" prints the value of that object. If a class had a member named `data`, you could use `this->data` to refer to it. However, this is bad style: anywhere it is legal to say `this->data` it means the same thing to say, simply, `data`. There is never a need to write the `this->` in front of a member name.

In this chapter, we use "`this`" three times in the Van program (lines 25 and 86), once in the Layers program (line 129) and once in the Copy program (line 225). These lines show correct usage of `this` and are repeated below:

```
(line 25)  out <<"    Box @ " <<this <<" : " <<*this;
(line 86)  out <<"  Van @ " <<this <<"  Load @ " <<&Load <<".\n";
(line 129) out << "Instance is: " <<this <<" name: " <<name <<endl
(line 225) return *this;
```

In lines 25 and 86, we wish to print the memory address of the implied parameter, so we write `<< this`. At the end of line 25, we wish to print the object itself, not its address, so we write `*this`. Line 225 returns a `Copy&`, which is a reference to the implied parameter.

10.1.2 Constructor Initializers

In C and C++, initialization follows rules that are more permissive than the rules for assignment. Anything written in the body of a constructor function follows the rules for assignment. If an action can only be done with initialization, it must be done using a ctor (constructor-initializer).

The term "constructor initializer" is shortened to "ctor". Ctors exist to provide a way to initialize an object during construction, rather than by using assignment later, in the body of the constructor function. The initial value for any non-static data member may be supplied this way and ctors are often used if the initial value is a parameter to the constructor. To illustrate ctor syntax, the constructor for the Stack class (Chapter 4) is repeated, below, then rewritten using three ctors. In this case, the choice is a matter of style.

```
Stack::Stack( cstring label, int sz )  {
    s = new char[max=sz];
    top = 0;
    name = label;
}

Stack::Stack(char* label, int sz): max(sz), top(0), s(new char[sz]), name(label){}
```

Why use ctors? Ctors provide a compact and easy way to initialize the data members of any object, but they are actually essential in three situations:

- A ctor is required if a class member is declared as const but optional for member variables.
- An object can compose other objects. Sometimes the constructor for the composed object has parameters; these must be supplied by a ctor.
- When class derivation is used, an object (the derived object) may be an extension of another object (the base object). If the constructor for the base class needs arguments, they must be supplied by a ctor.

What is the syntax? How you write a ctor depends on the type of the member it will initialize.

- If a class member is a non-class type, the name of the member is written followed by its initial value in parentheses. The initial value follows the same syntactic rules as an initializer in a declaration.
- If a data member is a class type, you must supply arguments for the constructor of the component class. To do so, write the member name followed by parentheses that enclose the name of its class and arguments for its constructor.
- To initialize the base portion of an object in a derived class, the name of the base class is written followed by the arguments for its constructor, in parentheses.

Where do you write them? The ctors are written before the beginning of the body of a constructor, between the closing parenthesis and the opening curly bracket. The list of ctors is preceded by a colon and separated by commas. Each ctor is used to initialize one data member of the class. If there is more than one ctor, they should be listed in the same order as the members are listed in the class declaration.

10.2 Dynamic Allocation Incurs Overhead.

The allocation functions “malloc” and “new” must calculate and store bookkeeping information for later use by “free” or “delete”. In this discussion, we will use the declarations below (an Item* initialized to a dynamically allocated Item) to illustrate the way things work. Figure 10.1 shows how the allocation area is configured on many systems.

The memory management system needs to know the block length. In both C and C++, the system must know how many bytes are in each dynamically allocated storage block. This could be stored as part of the block, computed from the sizeof the base type, or kept in a hidden table, indexed by the first address of the block.

```
struct Item { char name[20]; int count; float price; };
Item* purchase = new Item;
Item* inventory = new Item[5];
```

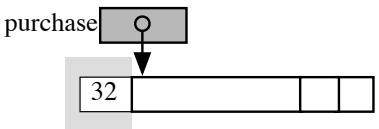

Figure 10.1: Allocation overhead.

There is no rule in the language standard that the system must allocate a particular amount of overhead space, or where it must be. However, every compiler must do something of this sort. Here we illustrate an old

and common strategy, which was used for both C and C++ for many years. Extra space was allocated at the beginning of every dynamically allocated area and initialized to the actual number of bytes in the allocation block, including the overhead. In Figure 10.1, this length field is shown against a pale gray background. Note that this field precedes the user's storage area, and the address returned by `malloc()` was the address of the beginning of the user's area.

A C++ system needs to know the number of slots in an array. When "new" is used to allocate an array of class objects, it stores the number of allocated array slots immediately before slot 0. This number is used by "delete[]" to loop through the array elements and call the appropriate destructor to recycle the extensions of each element before recycling the array itself. Figure 10.2 is a diagram of the actual storage allocated for the dynamic array on the third line of code, above. The array length is shown against a pale gray background.

If, in addition, the allocation length is stored with the array, it would be stored before the number of array elements. The remaining diagrams will show the array length but not the overall block length.

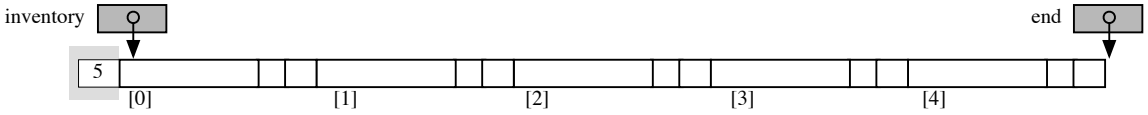

Figure 10.2: Overhead for an array of class objects.

10.2.1 Allocation Example: a Van Containing Boxes

The following short demonstration program will be used throughout this chapter to illustrate the allocation and deallocation semantics of C++. The first portion consists of a main program and two classes, Box and Van (which contains an array of Boxes). A second example builds on these two classes, introducing a third class named Layers that composes Van and Box. The memory diagrams shown are taken from actual runs of the program compiled by Gnu C++ for OS-X.

The Box class.

```
1   //-------------------------------------------------------------------------
2   // A. Fischer October 9, 2000                                  file: box.hpp
3   //-------------------------------------------------------------------------
4   #pragma once;
5   #include "tools.hpp"
6   //-------------------------------------------------------------------------
7   class Box {
8     private:
9       int length, width, high;   // The 3 dimensions of a box; occupies 12 bytes.
10
11    public:
12      Box(int ln, int wd, int ht){
13          length=ln; width=wd; high=ht; cout<<"\n  Real Box ";
14      }
15      Box(){ length = width = high = 1; cout <<"Default Box ";}
16
17      ~Box() {  cerr<< "  Deleting: ";  dump( cerr ); }
18
19      ostream& print( ostream& out ){
20          return out <<length <<" by " <<width <<" by " <<high <<".  ";
21      }
22      inline void dump( ostream& out );
23  };
24  //-------------------------------------------------------------------------
25  inline ostream& operator<< (ostream& out, Box& B) { return B.print( out ); }
26  inline void Box::dump(ostream& out){ out <<"  Box @ " <<this <<" : " <<*this <<endl;}
```

1. In order to declare an array of class objects, the class must have a default constructor, that is, one that can be called without parameters. This class has two constructors: a default constructor (line 15) for

initializing the array and a real constructor (lines 12–14) for building data objects. By using default parameters, these two definitions could be combined into one:

```
Box (int ln=1, int wd=1, int ht=1) { length=ln; width=wd; high=ht; }
```

2. A *dump* is a kind of debugging output that shows the machine addresses of your objects as well as their values. Dumps are helpful in C++ to verify what is happening with allocation, copying, and deallocation. This class contains a Dump function (line 22 and line 26).

3. Box::Dump() uses the keyword "`this`" to access its own address and print out its own contents.

4. `operator<<` must be defined before Box::Dump() because Dump() uses `<<` to print a Box.

The Van class.

```
27  //-------------------------------------------------------------------------
28  // Van is a container class that can hold a load of Boxes.
29  // A. Fischer October 9, 2000                                  file: van.hpp
30  // Hex addresses from a run on OS-X, March 1, 2009
31  //-------------------------------------------------------------------------
32  #pragma once;
33  #include "tools.hpp"
34  #include "box.hpp"
35  //-------------------------------------------------------------------------
36  class Van {
37    private:
38      int Count;      // @ f718, 4 bytes
39      int Max;        // @ f71c, 4 bytes
40      Box* Load1;     // @ f720, 4 bytes -> 1001b4, 0x40 = 48 bytes: 12*3 + 4 + ?
41      Box* Load2;     // @ f724, 4 bytes -> 1001f4
42
43    public:
44      Van (int n=6);
45      ~Van();
46      ostream& print( ostream& out );
47      void dump( ostream& out );
48      int load_dim(){ return *( (int*)(&Load1) - 1 ); }
49  };
50  //------------------------------------------------------------ Global declaration
51  inline ostream& operator<< (ostream& out, Van& V) { return V.print( out ); }

52  //-------------------------------------------------------------------------------
53  // Array construction example: Class implementation for Van.
54  // A. Fischer October 4, 2000                                  file: van.cpp
55  //-------------------------------------------------------------------------------
56  #include "van.hpp"
57  Van::Van (int n){   //-----------------------------------------------------
58      int a, b, c;
59      Count = 0;
60      Load1 = new Box[Max=n];
61      Load2 = new Box[2];
62
63      cout <<"\nEnter the boxes; use a 0 dimension to quit\n";
64      for (Count=0; Count<Max; ++Count) {
65          cout <<"  Dimensions: ";
66          cin >> a >> b >> c;
67          if (a*b*c == 0) break;
68          Load1[Count] = Box(a, b, c); // Make a local box, copy it using assignment.
69      }                                // Delete the local box at end of loop body.
70  }
71
72  Van::~Van(){    //-----------------------------------------------------------------
73      delete[] Load2;                    // Delete the boxes in the Load array.
```

```
74      cerr <<"  Deleted Load2.\n";
75      delete[] Load1;                    // Delete the boxes in the Load array.
76      cerr <<"  Deleted Load1.\n";
77  }
78
79  ostream& Van::print( ostream& out ){    //---------------------------------------
80      out <<"Load 1 has " <<Count <<" Boxes.\n";
81      for (int k=0; k<Count; ++k)  out <<"        " <<Load1[k];
82      return out << endl;
83  }
84
85  void Van::dump( ostream& out ) {        //---------------------------------------
86      out <<"  Van @ " <<this
87          <<"\n  Count @ " <<&Count <<" = " << Count
88          <<"\n  Load2 @ " << &Load2 <<" = " << Load2<<".\n"
89          <<"\n  Load1 @ " << &Load1 <<" = " << Load1
90          <<" slots " << load_dim() <<endl;
91      for (int k=0; k<Count; ++k)  Load1[k].dump( out );
92      out << endl;
93  }
```

Only two comments are given here. Most of the commentary on this class is in the next section, where allocation, initialization, and deallotion actions are examined in detail.

1. The Van constructor (lines 44 and 57–70) has a default parameter. This lets us declare a Van with or without parentheses and a dimension following the name, as in "`Van V1`" or "`Van V2(10)`".

2. Van::Dump() calls the function defined at the bottom of van.hpp (line 48). This functions accesses the hidden dimension field that precedes the beginning of the array named `Load1`. After printing this information, Van::Dump() calls Box::Dump() in a loop (line 91), to dump the Boxes in the van.

A test run on OS-X.

```
A dynamic array of class objects.
Default Box Default Box Default Box Default Box Default Box Default Box Default Box
Enter the boxes; use a 0 dimension to quit
  Dimensions: 2 2 2

  Deleting:   Box @ 0xbffff328 : 2 by 2 by 2.
  Real Box   Dimensions: 3 3 3

  Deleting:   Box @ 0xbffff328 : 3 by 3 by 3.
  Real Box   Dimensions: 4 3 2

  Deleting:   Box @ 0xbffff328 : 4 by 3 by 2.
  Real Box   Dimensions: 0 0 0

About to dump the van:
  Van @ 0xbffff3d8
  Count @ 0xbffff3d8 = 3
  Load1 @ 0xbffff3e0 = 0x500184 slots 5 length 3
  Load2 @ 0xbffff3e4 = 0x5001c4.
  Box @ 0x500184 : 2 by 2 by 2.
  Box @ 0x500190 : 3 by 3 by 3.
  Box @ 0x50019c : 4 by 3 by 2.


Normal termination.
  Deleting:   Box @ 0x5001d0 : 1 by 1 by 1.
  Deleting:   Box @ 0x5001c4 : 1 by 1 by 1.
  Deleted Load2.
  Deleting:   Box @ 0x5001b4 : 1 by 1 by 1.
  Deleting:   Box @ 0x5001a8 : 1 by 1 by 1.
  Deleting:   Box @ 0x50019c : 4 by 3 by 2.
  Deleting:   Box @ 0x500190 : 3 by 3 by 3.
  Deleting:   Box @ 0x500184 : 2 by 2 by 2.
  Deleted Load1.
```

The main program.

```
 94  //-------------------------------------------------------------------------------
 95  // Array construction example, October 8, 2000                    file: vanM.cpp
 96  //-------------------------------------------------------------------------------
 97  #include "tools.hpp"
 98  #include "van.hpp"
 99  //-------------------------------------------------------------------------------
100  int main( void )
101  {   cout << "\nA dynamic array of class objects.\n";
102      Van V;                                  // Make a Van with space for 5 Boxes.
103      // Van V(2);                            // Alternate version has 2 Boxes.
104      cout <<"\nAbout to dump the van:\n";
105      V.dump( cout );
106      bye();
107  }
```

10.2.2 How Allocation and Deallocation Work

Consider the Van program above, which declares a Van named V (line 102). In response, the system allocates space for the four core data members of a Van (dark gray in Figure 10 .3) and calls the Van constructor to initialize that space.

Construction of the Van. The Van constructor allocates space for an array of five Boxes and stores the pointer in V.Load. Automatically, the default constructor for Boxes is run five times, once for each array slot. The five Box objects are initialized by the default Box constructor, so the dimensions of all five are initialized to 1 1 1. Figure 10 .3 illustrates the actual storage that was allocated for this object on my machine, with the initial values from the default constructor. The core portion of the object "V" is colored dark gray. The extensions consist of overhead (light gray) and usable storage (white).

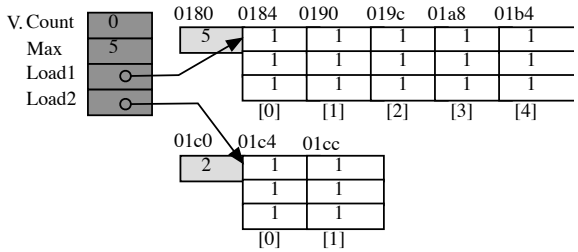

Figure 10.3: Creating a Van full of Default Boxes

Storing Boxes in the Van. Then control passes to the code part of the Van constructor. Line 63 prints instructions and the following loop prompts the user for the dimensions of a series of boxes. For each set of dimensions entered, line 68 constructs a new temporary Box and copies it into one slot of the array of Boxes. This replaces one set of dummy data in the Van by real information. The default assignment operator is used for this purpose.

```
Load1[Count] = Box(a, b, c);
```

When control reaches the end of the loop body (line 69), the temporary object is automatically deleted and we see a deletion comment. In the run shown above, data for three boxes were entered, followed by a dummy box with a volume of 0, which ended the input phase. Figure 10.4 illustrates the contents of the Van at the end of the input phase.

By the end of the input loop, ten Boxes have been allocated and seven still exist:

- Five are part of Load1.
- Two are part of Load2.
- Three were allocated, initialized, copied into Load1, and freed.

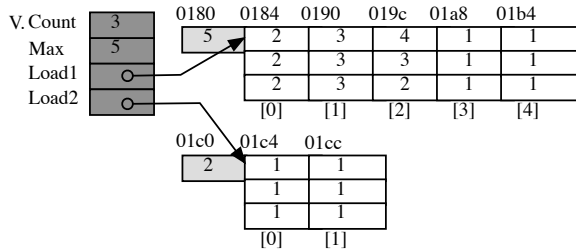

Figure 10.4: After storing real Box data in the Van.

As proof of this deletion, we see three trace comments:

```
Deleting:   Box @ 0xbffff328 : 2 by 2 by 2.
Deleting:   Box @ 0xbffff328 : 3 by 3 by 3.
Deleting:   Box @ 0xbffff328 : 4 by 3 by 2.
```

Note that the addresses of these three boxes are the same! that happens because the first one is created, used, and deleted, then the second is created in the same memory location. Also note that these boxes were allocated at an address that is very distant from the Van boxes (shown at the end of the output). This difference happens because the temporary Boxes are allocated on the run-time stack while the Van Boxes are allocated in the dynamic storage area.

Dumping the Van. Line 105 of main() dumps the Van, producing the second block of output. This block shows the machine addresses of the Van and its core parts, as well as the addresses of its two dynamically allocated extensions. From the dump, we see that the data from the temporary boxes has, indeed, been copied into the permanent boxes that are attached to the van.

Deallocation of the Van with delete[ ]. When we wish to delete dynamic storage, we use the name of the variable that points at the dynamically allocated block. Assume this pointer is named X. There are two forms of the deallocation command: `delete X` and `delete[] X`. The conservative rule for usage is:

> If you allocate with [], use delete[] with [].

Different treatment is necessary for arrays and non-arrays. For a non-array, the only memory space that needs deletion is the single variable that X points at. But for an array, we need to deal with the array itself *and* with the objects stored in it. Allocating the array caused the constructors to be run for the objects in the array, so deleting the array must cause the destructor to be run for each individual array object.

In the Van program, the statements that create the dynamic storage are on lines 60 and 61: `Load1 = new Box[Max = n];` and `Load2 = new Box[2];` Since we allocate this array in the Van constructor, and since this statement stores the allocation pointers in Load1 and Load2, we write `delete[] Load1` and `delete[] Load2` in the Van destructor. The result is:

- The Box destructor will be run for each Box in the array, in the opposite order from their creation. The number that is stored just before the `&Load1[0]` will control the deletion loop.
- Last of all, the array, itself, will be deallocated.

The last line of `main()` is the call on `bye()`, which prints the "Normal termination" message. The program's work is done and it is time for it to return to its caller (the system). We say that the variables in main() *go out-of-scope*, which causes the destructors for main's variables to be run. (`Van::~Van` is run to delete V.) You do not need to write anything to cause this to happen, and there is no way to prevent it. In our test run, the third block of output shown is the result of the automatic destruction. First, the Van destructor was called by the system. On line 73, it called the Box destructor: `delete[] Load2`. When the 2 Boxes have been deleted, control returns to the Van desctuctor. In the output, we see the trace comments from deleting two boxes, followed by the trace for deleting the array. On line 75, the Box destructor is called again: `delete[] Load1`. When all the Boxes have been deleted, control again returns to the Van desctuctor. In the output, we see the trace comments from deleting five boxes, followed by the trace for deleting the array.

10.3 Construction of C++ Objects

10.3.1 A Class Object is Constructed in Layers

- Space is allocated for the core portion of the whole object.
- Starting with the first component member, the constructor functions for the members are called. These initialize the components and may cause extensions to be allocated and initialized. If any of these constructors have parameters, they must be passed as ctor initializers (see below).
- Members of the class are initialized from ctor initializers, if any.
- The constructor for the class, itself, is called. It may store values in components and may also create extensions.

10.3.2 Layering Demo Program

The goal of the Layers program is to illustrate the order in which aggregate objects are constructed and deleted. The trace comments in the Layers functions, together with those in Van and Box provide step-by-step evidence of how C++ storage management works. The Layers class is given next, followed by comments that lead you through the step-by-step construction process and, finally, the output.

The Layers class.

```
110  //-----------------------------------------------------------------------------
111  // The Layers class aggregates Van and Box.
112  // A. Fischer October 10, 2000                                  file: layers.hpp
113  //-----------------------------------------------------------------------------
114  #pragma once;
115  #include "van.hpp"
116
117  class Layers {      //-------------------------------------------------------------
118      char* name; // Memory is allocated by body of Layers constructor.
119      Box B;
120      Van V;
121
122   public:            //-------------------------------------------------------------
123      Layers( char* nm ) : B(8,7,6), V(4) {               // constructor with ctors
124          cout <<"  Constructing " <<nm <<"\n";
125          name = new char[strlen(nm) + 1];
126          strcpy(name, nm);
127          cout <<"  " <<nm <<" is complete.\n\n";
128      }
129
130      ~Layers() {                                        // destructor
131          cout << "  Deleting name @ " <<(unsigned)name <<": " <<name << endl;
132          delete name;
133      }
134
135      void print( ostream& out ){
136          out << "Instance is: " <<this <<" name: " <<name <<endl
137              <<"   Box: " <<B <<"\n   Van: " <<V;
138      }
139  };
```

- An object of class Layers has three members: a char*, a Box, and a Van.
- The Layers constructor has a char* parameter; this name will be used in the output to help clarify which object is being constructed or deleted. Most of the output, below, came from the class constructors and destructors.
- Line 123 uses two ctors to initialize the Box and Van portions of the object. This is necessary because we want a real box, not a dummy, and we want a Van with 3 slots, not 5 (the default).

- A cast is used in the destructor to print the address of the name string (line 131). Without the cast, we would print the word itself. The contents of `name` is the address of the dynamic allocation area (a pointer). We are allowed to cast pointers to type unsigned.
- The last part of the trace, after the "normal termination" comment, shows that six Boxes were created as part of the Layers object created when line 149 of main was executed. The first box created was the loose box declared on line 119 and initialized by a ctor in the Layers constructor on line 123. The ctor supplies initialization paramters, so this box was initialized by the "Real" Box constructor, the one with parameters.
- The other five Boxes are inside the Van that is declared on line 120 and initialized by the second ctor on line 147. The Van constructor is on lines 57...70. On lines 60 and 61, the constructor creates two new arrays of Boxes. Since these boxes are in an array, they are initialized by the default constructor.
- The first half of the output and shows the user-dialog that occurred during the process of constructing three real boxes and storing them in the Van, lines 63...69 in the Van constructor. The Box objects created there are local temporary variables that are deallocated when control reaches line 70. These variables exist just long enough to copy their contents into the blank box that is part of the Van array (using the default definition of the assignment operator, which is the same as the default copy constructor).
- The parts of an object are initialized before calling the constructor for the whole. Note that control enters the Van constructor (output line 170) after initializing all the Boxes to default values. It enters the Layers constructor and prints the object's name, "PackRat", after initializing the Van.
- The main program creates an instance of class Layers (line 149) and prints it. From the output, you can see that the Layers object contains a Box and a Van, and the Van contains two more real Boxes (the default boxes are not printed here).
- The post-termination output verifies the order in which objects are deleted: the destructor for the outside class triggers the destructors for the inside classes before it executes itself. Within an array, as in Van.Load1, the array elements with the largest subscripts are deleted first and the object in slot 0 is deleted last. Then the array itself is deleted.

The main Layers program.

```
140  //-------------------------------------------------------------------------------
141  // Layered construction and destruction example.
142  // A. Fischer October 10, 2000                                  file: layersM.cpp
143  //-------------------------------------------------------------------------------
144  #include "tools.hpp"
145  #include "layers.hpp"
146  //-------------------------------------------------------------------------------
147  int main( void ) {
148      cout << "Creating an object of class Layers. ";
149      Layers f1( "PackRat" );
150      f1.print(cout);
151      // Layers f2(f1);        // A call on the copy constructor.
152      bye();
153  }
```

The trace.

```
Creating an object of class Layers.
Real Box Default Box Default Box Default Box Default Box Default Box
Enter the boxes; use a 0 dimension to quit
Dimensions: 3 2 3

Deleting:   Box @ 0xbffff624 : 3 by 2 by 3.
Real Box   Dimensions: 2 2 2

Deleting:   Box @ 0xbffff624 : 2 by 2 by 2.
Real Box   Dimensions: 4 2 1

Deleting:   Box @ 0xbffff624 : 4 by 2 by 1.
```

```
Real Box   Constructing PackRat
PackRat is complete.

Instance is: 0xbffff708 name: PackRat
Box: 8 by 7 by 6.
Van: Load 1 has 3 Boxes.
3 by 2 by 3.          2 by 2 by 2.          4 by 2 by 1.

Normal termination.
Deleting name @ 1049120: PackRat
Deleting:   Box @ 0x1001c0 : 1 by 1 by 1.
Deleting:   Box @ 0x1001b4 : 1 by 1 by 1.
Deleted Load2.
Deleting:   Box @ 0x10019c : 4 by 2 by 1.
Deleting:   Box @ 0x100190 : 2 by 2 by 2.
Deleting:   Box @ 0x100184 : 3 by 2 by 3.
Deleted Load1.
Deleting:   Box @ 0xbffff70c : 8 by 7 by 6.
```

10.4 Constructors and Construction

C++ objects are created and initialized in three situations:

1. A call on `new`
2. Declaration of a class-type variable.
3. A call on a constructor function embedded in an expression.

In Java, the the first way listed is the only way to create non-array objects. n both languages, an object created with `new` is allocated in the dynamic memory area called the "heap". These objects are used through references (in Java) or pointers (in C++). Because they are in the heap, they can outlast the execution of the function that created them and the lifetime of the variables that store them. In C++, these objects must be specifically deallocated by calling `delete` when they are no longer needed.

Unlike Java, C++ also permits local objects to be created by a declaration or by calling any constructor. These objects are allocated in the current stack frame and deallocated when they go "out of scope". (That is, when control leaves the block of code that contains the declaration or constructor call.)

Kinds of Constructors The terminology regarding constructors is confusing. However, the types of constructors and the various terms for them must be mastered before you will understand C++ reference books. The types of constructors are:

1. **Constructors with parameters.** This is the usual case and is the kind we have been using. Examples are found in the Van program on lines 14, 44 / 60–73, and are in the Layers program on lines 146–151. These constructors can be used in declarations (lines 106 and 129), ctors (line 146), and calls on new that allocate a single object, not an array. Line 71 calls a constructor in an assignment statement. This builds a temporary object that will be discarded as soon as its value is copied into the array.
2. **Null constructor.** A null constructor is one that initializes nothing. A null constructor for a class named named `Zilch` looks like this:

   ```
   Zilch(){}
   ```

   Sometimes we add trace output to constructors; these are still null constructors, even though the function body is non-blank. A null constructor with a trace might look like this:

   ```
   Zilch(){ cerr << "Constructing a Zilch object."; }
   ```

   Every class has a constructor. If you don't provide any constructor at all, C++ generates a null constructor. If you provide some sort of constructor, a null constructor will not be supplied automatically, although you can define one yourself.

3. **Default constructor.** A default constructor is a one with no parameters (line 15) or one that has a default value for every parameter (line 44 / 60–72). A null constructor is a default constructor, but non-null constructors constructors can also be default constructors. A declaration that uses the default Van constructor is on line 105.

   If you declare an array of class objects, the class *must* have a default constructor which will be used to initialize the elements. The calls on new on lines 63–64 use the default Box constructor to initialize all the boxes.

4. **Default copy constructor.** A copy constructor is used whenever there is a need to initialize a newly constructed object from another object of the same class. The most common and important application is call-by-value parameter passing, but uses of the copy constructor are also shown on lines 71 and 130. Each class has exactly one copy constructor (either by default or by explicit definition) with one of these prototypes:

   ```
   Classname( const Classname& );
   Classname( Classname& );
   ```

   A default copy constructor is supplied automatically if no explicit copy constructor is present. It performs a shallow copy from the initializing object into the newly-constructed object. If a user-defined copy constructor is present, it is used instead of the default copy constructor for call-by-value. If `operator=` is not redefined for a particular class C, then the default copy constructor is used to implement `c1 = c2`, where c1 and c2 are objects of class C.

5. **A deep-copy constructor.** A deep copy is a complete copy of a data structure, including its extensions. A programmer-defined copy constructor that performs a deep copy is shown on Lines 220. . . 224 of the Ball program (next). Calls on this copy constructor are made on lines 200. . . 201.

   If a data structure structure is small, the time and space used to create this copy are often not important. However, if the data structure is large, making a deep copy for every call on every function would waste an excessive amount of time and space and would cause an unacceptable performance slowdown. For this reason, deep-copy constructors are rarely used.

10.4.1 A Variety of Constructors

This section introduces a class named Ball with four constructors and a redefinition of the assignment operator. The main program calls each one to illustrate the differences in syntax.

Calling the constructors:

1. A static class member is shared by all instances of the class. In this program, we use a static member to count the total number of instances that have been created. This static member is created and initialized on line 306.
2. The ID field of each instance is the value of the counter at the time the instance was initialized. This counter is incremented by every constructor except the null constructor which has been commented out.
3. The declaration on line 311 has just the class name and an object name; this calls the constructor on lines 345–49 and uses the default value for the parameter. Line 378 of the output shows that the Name field has been initialized to the default value, 'B', the ASCII version of the argument 66.
4. Line 312 calls the same constructor as line 311 but supplies the optional parameter. The result is shown on line 379: the Name field has been initialized to 'K', rather than to the default, 'B'.
5. The declaration on line 313 provides a char* argument for the constructor; this calls the normal constructor (lines 350–54). Line 380 of the output shows that the Name field has been initialized to the argument provided in the declaration.

```
300  // ------------------------------------------------------------------------
301  //  Syntax and semantics for copy constructors, initialization, assignment.
302  //  Alice E. Fischer May 6, 2001                          file: ballM.hpp
303  // ------------------------------------------------------------------------
304  #include "tools.hpp"
305  #include "ball.hpp"
```

```
306  int Ball::Counter = 0;     // Create & initialize the static class member.
307
308  // ------------------------------------------------------------------------
309  int main( void ) {
310      cout <<"                                    ID  Name\n";
311      Ball One;           cerr <<"a. " << One <<endl;     // Default constructor
312      Ball Two( 75 );     cerr <<"b. " << Two <<endl;     // Normal constructor
313      Ball Three("Ali");  cerr <<"c. " << Three <<endl;   // Normal constructor
314      Ball Four( One );   cerr <<"d. " << Four <<endl;    // Copy constructor
315      Ball Five = Three;  cerr <<"e. " << Five <<endl;    // Copy Constructor
316      cout << "\nReady to construct an array:" <<"\n";
317      Ball Six[2];        cerr <<"\t" <<"f. "<< Six[0] <<"\t   " <<Six[1] <<"\n";
318      // Ball Bad = Ball(77);                    // Bad: Ball(77) not a Ball&
319      cerr <<"\nNow " << One.getcount() <<" Ball objects have been created.\n";
320      Four = Two;         cerr << "g.........................\n" // Assignment
321                               <<"\t" << Four <<"\n\t"  << Two <<endl;
322      bye();
323      return 0;
324  }
```

6 The argument in the declaration on line 314 is an object of class Ball; this calls the copy constructor (lines 355-59). This copy constructor gives the new object a unique ID#, then makes a deep copy of the rest of the argument. It does not change the argument. Within this function, the parameter name is used to refer to the object being copied and the corresponding fields of the new object are called by the member name alone. Line 381 of the output shows the result: the Name field has been initialized to "B", a copy of the Name in One.

7 Line 315 uses another syntax to call the copy constructor, initializing the new object, Five, as a copy of Three. The result is on line 382.

8 At this time, five objects have been created and intialized. We now create an array of two more Balls, on line 317, giving us a total of seven Ball objects. The output (lines 384–87) shows that the constructor on lines 345–49 was used, with the default value of the parameter, to construct both.

9 We now use the redefined assignment operator (line 320 / 360–64). This is a very strange definition for `operator=` because it only copies one character from the right operand into the left operand, but it does illustrate how redefinition works. Initially, Four.Name was "B". After executing line 362, Four.Name has been changed to "K", as shown on line 390.

10 We have reached the end of the program; the call on `bye()` prints a termination comment (lines 322 / 393). After that, we see the trace comments printed by the destructor on line 345. As each object is deleted, we decrement the object counter. The last line of the trace tells us that the Ball class does not have a memory leak.

11 A class cannot have two functions with the same calling sequence. The null constructor on line 339 is commented out because it conflicts with the default constructor on lines 345–49. When both are in the code, the compiler produces comments like this:

```
ballM.cpp: In function ‘int main ()’:
ballM.cpp:12: call of overloaded ‘Ball()’ is ambiguous
ball.hpp:17: candidates are: Ball::Ball ()
ball.hpp:27:                 Ball::Ball (int = 66)
```

In this program, we fix the problem by commenting out the null constructor. If, instead, we comment out the default constructor on line 345...349 (and the call on line 312), we see output from the null constructor:

```
Null constructor........a. 782336
```

The ID and Name fields contain garbage because they have not been initialized.

12 The system's final status comment (line 402) assures us that the Ball program' s memory has been freed without crashing.

The Ball class.

```
// ------------------------------------------------------------------------
//  Syntax and semantics for copy constructors, initialization, assignment.
//  Alice E. Fischer May 6, 2001                              file: ball.hpp
// ------------------------------------------------------------------------
#pragma once;
class Ball {    // ---------------------------------------------------------
  private:
    static int Counter;
    const int ID;
    char* Name;

  public:       // ---------------------------------------------------------
    //Ball(){ cerr<< "Null constructor........"; }  // Null constructor
    //~Ball(){}                                     // Null destructor.
    ~Ball(){
        --Counter;  delete Name;
        cerr <<"\tDestructor..." << Counter <<" Balls remain.\n";
    }
    Ball( int alpha = 66 ): ID( ++Counter ) {  // --- Default constructor.
        cerr <<"Default or normal constructor. ";
        Name = new char[ 2 ];
        Name[0] = alpha; Name[1] = '\0';
    }
    Ball( char* arg ): ID( ++Counter ) {    // -------- Normal constructor.
        cerr <<"Normal constructor.............";
        Name = new char[ 1+strlen(arg) ];
        strcpy( Name, arg );
    }
    Ball( Ball& a ): ID( ++Counter ) { // ---------- Deep copy constructor.
        cerr <<"Copy constructor...............";
        Name = new char[ 1+ strlen(a.Name) ];
        strcpy( Name, a.Name );
    }
    Ball& operator = ( Ball& source ){ // -- Changes data but not identity.
        cerr<<"Assignment function...........\n ";
        Name[0] = source.Name[0];
        return *this;
    }

    void print() { cerr <<ID <<"  " <<Name ; }
    int getcount() { return Counter; }
};
ostream& operator <<( ostream& out, Ball& C ){ C.print(); return out; }
```

The output:

```
                                ID  Name
Default or normal constructor. a. 1  B
Default or normal constructor. b. 2  K
Normal constructor.............c. 3  Ali
Copy constructor...............d. 4  B
Copy constructor...............e. 5  Ali

Ready to construct an array:
Default or normal constructor. Default or normal constructor.   f. 6  B    7  B

Now 7 Ball objects have been created.
Assignment function...........
 g.........................
    4  K
    2  K
```

```
392
393  Normal termination.
394      Destructor...6 Balls remain.
395      Destructor...5 Balls remain.
396      Destructor...4 Balls remain.
397      Destructor...3 Balls remain.
398      Destructor...2 Balls remain.
399      Destructor...1 Balls remain.
400      Destructor...0 Balls remain.
401
402  The Debugger has exited with status 0.
```

10.5 Destruction Problems

Deep deletion after a shallow copy When call-by-value is used to pass an argument to a function, the class copy constructor (default or user-defined) is used to initialize the parameter variable with a copy of the argument value. The default copy constructor does a shallow copy.

At function-return time, the class's destructor will be run on the parameter. A correctly-written destructor deallocates object extensions and frees dynamic memory (deep destruction). However, when shallow copy is used to pass an argument, the parameter variable and the underlying argument variable both point to the same object extensions. Using deep destruction will cause a serious problem because the object extensions will be deleted. However, the calling program still needs them.

Therefore, call-by-value generally cannot be used with parameters of class types. Use call by reference (more efficient, less modular) or define a copy constructor that does a deep copy (miserable efficiency, great modularity). The next sample program shows a class with a copy constructor that makes a deep copy.

Important reminders.

- Deletion of stack objects happens when control leaves the block in which the object was declared.
- Deletion of dynamically-allocated (new) objects happens when delete or delete[] is called explicitly. An attempt to use delete on a stack-allocated object is a fatal error.
- In both cases, the destructor from the appropriate class is called to deallocate memory.
- Memory leaks happen when you fail to delete a dynamic object at the end of its useful life, before the last pointer to that object is reassigned to point at something else.
- Fatal (but subtle and delayed) errors happen when you attempt to use an object after it has been deleted, when you delete the same thing twice, or when you delete something that was not created by new.