Chapter 6: Objects, Allocation, and Disasters

By analyzing errors, we can learn how to make programs function well:

> **Knowledge rests not upon truth alone, but upon error also.**
> —Carl Jung, one of the fathers of modern psychotherapy.

6.1 Objects: Static, Automatic, Dynamic, and Mixed

6.1.1 Storage Class

- Auto objects are created by variable and parameter declarations. An object is allocated in the stack frame for the current block when a declaration is executed and deallocated when control leaves that block.
- Static objects are created by variable declarations with the "static" qualifier. These are allocated and initialized at load time and exist until the program terminates. They are only visible within the function block that declares them. The word "static" comes from "stay". Static variables stay allocated and stay in the same memory address, even while control leaves and reenters their defining block many times.
- Dynamic objects are created by explicit calls on `new`. They continue to exist until they are deleted or until the program ends. All objects that contain dynamic memory also have an auto or static portion (the core of the object) which provides access to the dynamic portion or portions (the extensions of the object).

It is common for a single object to have members of two or three storage classes. In Figure 6.1.1, the class declaration creates a simple class that is partly auto and partly dynamic, with an appropriate constructor and destructor. The diagram to the right of the declaration shows the object that would be created by the declaration `Mixed A( "Joe" )`. The core portion is shown against a shaded background, the extensions against a white background.

```
class Mixed {
   int len;
   char* name;
public:
   Mixed( char* nm ) {
      len = strlen( nm );
      name = new char[ len+1 ];
      strcpy( name, nm );
   }
   ~Mixed() { delete[] name; }
}
```

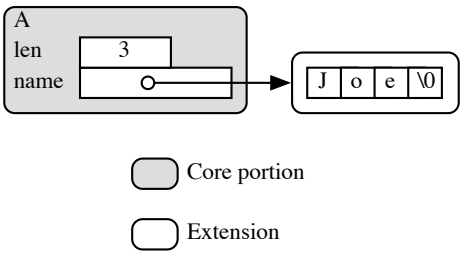

Figure 6.1: An object built partly of auto storage and partly of dynamic storage.

6.1.2 Assignment and Copying

The `=` operator is implicitly defined for all basic C++ types and for all types that the programmer might define. If `A` and `B` are of the same type, then `B = A;` is always defined and (by default) does a shallow copy.

- Shallow copy (left side of Figure 6.1.2): a component-wise copy of the non-dynamic portion of the object. This kind of copy rule is used to implement call-by-value, unless a different copying rule is explicitly defined.

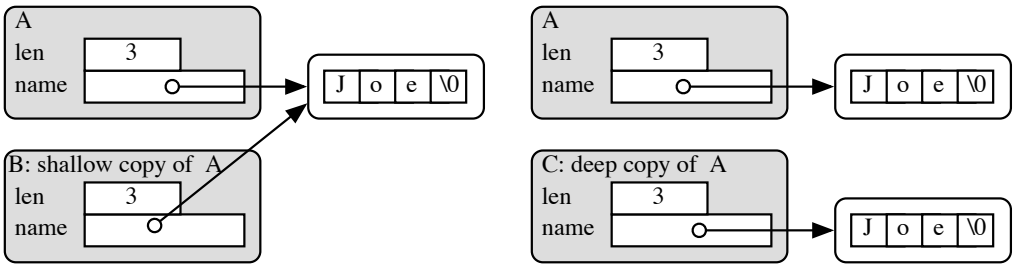

Figure 6.2: Shallow and deep copies. In the shallow copy, two objects share the same extension.

- Deep copy (right side of Figure 6.1.2): a component-wise copy of the `auto` portion of the object, except for fields that are pointers to dynamically allocated parts. These members are set to point at newly allocated and initialized replicas of the object's extensions. This is never predefined; a function must be written for the class that allocates, initializes, and copies parts appropriately, one at a time.

Copying problems. When an object, `A`, is shallow-copied into another object, `B`, the two objects share memory and a variety of nasty interactions can happen:

- If, before the copy, a member of `B` is a pointer to an extension, there will be a memory leak because that pointer will be overwritten by the corresponding member of `A`
- Changes to the data in the extensions of either object affect both.
- Deleting an extension of either object will damage the other. Ultimately, when destructors are run on the two objects, the first will delete properly, the second will cause a double-deletion error.

Because of these potential problems, shallow copy must be used carefully in C++. On the other hand, making a deep copy of a large data structure consumes both time and space; so making many deep copies can kill a program's performance.

6.2 Static Class Members

Static function members. The purpose of a static class function is to allow a client to use the expertise of a class before the first object of that class exists. This is often useful for giving instructions and for validation of the data necessary to create an object.

A static function is called using the name of its class and `::`. For example, suppose a class named Graph has two static functions, `instructions()` and `bool valid( int n )`. Calls on these static functions might look like this:

```
Graph::instructions();
if (Graph::valid( 15 )) ...
```

Since a static class function must be usable before the first class object is created, the code within the function cannot use any of the data members of that class. It can use parameters and global constants, however.

Static data members. The purpose of a static class variable is perfectly simple: it is used to communicate among or help define all class instances. A static data member is a shared variable that is logically part of all instances of the class (and all instances of all derived classes).

For example, consider a dynamically allocated matrix whose size is not known until run time. It is important to create all matrix rows the same length, but that length cannot be set using `#define`. A solution is to declare the array length as a private static class member. Then it can be set at run time, but it is visible only within the class. It can be used to construct all class instances, but only occupies space once, not space in every instance.

Sometimes a set of constants is associated with a class, for example, a Month class might have three associated constant arrays, an array of valid month abbreviations or names, and the number of days in each month. Since these are constant, it is undesirable to keep a copy inside every Month object, and undesirable to keep creating and initializing local arrays. The ideal solution is to make a set of shared (static) arrays within the class.

Declaration. To declare a static class member, you simply write "static" before the member name in the class declaration. Since the variable must be created at load time, it cannot be initialized by the class constructor. Also, since a static data member is shared by many class instances, it cannot be created by any one of them. Therefore, it is allocated and initialized at load time. The C++ designers (in their great wisdom) decided that static class members must be declared outside the class as well as within and initialized outside the class (unless they are integer variables or enum variables and are also const).

Here are some declarations that could be in a Month class. Since these are all arrays, we are not permitted to initialize them within the class declaration. The int constant can be initialized here.

```
// Subscript 1 corresponds to January.  Subscript 0 is unused.
static const int miy = 12;       // Number of months in a year.
static const int dim[13] ;       // Number of days in each month.
static char* mAbbrev[13];        // 3-letter abbreviations of the names.
```

Creation and initialization. Static data members are created and initialized at program-load time. Because of this, a separate statement must be written, outside the class definition, to declare and (optionally) initialize each static class member. The initializer, if present, may depend only on data that is known at compile time. Note that the defining declaration *does* contain the class name and *does not* contain the word "static":

```
int Matrix::columns;      // Will be initialized after execution begins.
int Customer::count = 0;  // Run time counter for instances of Customer.

const int Month::miy;     // Initialized within the class; not needed here.
const int Month::dim[13]  = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
char* Month::mAbbrev[13]  = {"---","Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};
```

The defining declaration for a static data member must be compiled exactly once in all the modules that comprise the program. Since header files are often included by more than one module, they are not an appropriate place for the defining declaration. modules that comprise the program. If a class has a .cpp le, the static initialization can be written at the head of that file. This is good because it keeps the information about class members in the class own le. However, some classes are entirely contained within the .hpp le and have no corresponding .cpp le. For such a class, we could make a le just for one line, or we could put the static initialization just before the beginning of the main program. This placement guarantees that the dening declaration will be included exactly once in the completed, linked program but it separates an important fact (the initial value) from the denition of the class.

If a class has a `.cpp` file, the static initialization can be written at the head of that file. This is good because it keeps the information about class members in the class' own file. However, some classes are entirely contained within the `.hpp` file and have no corresponding `.cpp` file. For such a class, we could make a file just for one line, or we could put the static initialization just before the beginning of the main program. This placement guarantees that the defining declaration will be included exactly once in the completed, linked program but it separates an important fact (the initial value) from the definition of the class.

Usage. Static does not mean the same thing as const; a static data member is used in the same way that a non-static class variable would be used. Above, we showed declarations for two static variables. In the Matrix class, the variable named `columns` will be initialized by the Matrix constructor and used to create all the rows of the Matrix. In the Customer class, the counter will be incremented every time a Customer is created and decremented every time one is deallocated. There is nothing in the code that uses a static member to indicate that it is static or special in any way.

6.3 Common Kinds of Failure

1. **Memory Management Error.** Errors in memory management come in three varieties, below. The first two errors typically cause a program to crash. The third will not cause immediate bad behavior, but a leaky program will gradually cause the performance of the whole system to deteriorate, as it demands more and more unnecessary memory to run.

   (a) Deleting memory too soon, when it is still in use. (Dangling pointer)
   (b) Deleting memory that was not dynamically allocated. (Wrongful deletion)

(c) Failure to deallocate memory when it is no longer useful. (Memory leak) C++ does not have an automatic garbage collector; it is the programmer's responsibility to deallocate dynamic storage when it is no longer useful. If the last pointer to a dynamic object is lost before it is deleted, we have a "memory leak". The memory locations occupied by that object are not recycled and are therefore no longer available to the program. Some programs run continuously for days or weeks. In such a program, a gradual memory leak can cause system performance to slowly degrade. When the swap space or paging space is full, the program will hang.

In a properly working system, memory leaks are recovered when the program ends. However, some common current systems do not free memory properly, and the memory is not recovered until someone reboots the system.

2. **Amnesia.** You allocated and initialized dynamic memory correctly. Immediately after doing so, the information stored there was correct and complete. However, later, part or all of that information is gone, even though you have not stored anything in the object since it was created.
3. **Bus error.** Your program crashes because you have tried to use a memory address that does not exist on your computer.
4. **Segmentation Fault.** This happens on multiprogrammed systems with memory protection. Your program crashes because you have tried to use a memory address that is not allocated to your process. It may be part of the system's memory space or it may belong to another process that is in memory.
5. **Throwing an exception.** The program is in a permanent wait state or an infinite loop.
6. **Waiting for Eternity.** The program is in a permanent wait state or an infinite loop.

6.4 Causes and Cures

6.4.1 A shallow copy disaster.

```
1   #include "tools.hpp"
2   class Mixed {
3       int len;
4       char* name;
5       public:
6       Mixed( char* nm ){  //------------------- Constructor
7           len = strlen(nm);
8           name = new char [len+1];
9           strcpy( name, nm );
10      }
11      ~Mixed() { delete[] name; }
12      void print(){ cout << name << " " << len <<'\t'; }
13      void change( char x ) { name[0] = x; }
14  };
15
16  int main( void )
17  {
18      Mixed A( "Joe" );
19      Mixed B( "Mary" );
20      cout << "Initial values A, B:\t";  A.print();  B.print();
21      B = A;
22      cout << "\nAfter copying B = A:\t";  A.print();  B.print();
23      B.change( 'T' );
24      cout << "\nAfter setting B.name[0] = 'T':\t";  A.print();  B.print();
25      cout <<endl;
26  }
```

```
Initial values A, B:    Joe 3   Mary 4
After copying B = A:    Joe 3   Joe 3
After setting B.name[0] = 'T':  Toe 3   Toe 3
*** malloc[2029]: Deallocation of a pointer not malloced: 0x61850; This could be a double free(),
or free() called with the middle of an allocated block; Try setting environment variable
MallocHelp to see tools to help debug
```

Cause. This short program creates two objects of class `Mixed`, then assigns one to the other. By default, a shallow copy operation is performed. An unusual function named `change()` is included in this class to illustrate the shared nature of the extension. No normal class would have a public function, like this, that permits private data to be corrupted. The output illustrates all three properties of a shallow copy operation. First, the allocation area that was originally attached to `B` is no longer attached to anything and has become a leak. Second, because `B` and `A` now share storage, assigning `'T'` to the first letter of `B` changes `A` also, as shown in the output, below. Finally, my run-time system detects a double-deletion error during the program termination process, and displays an error comment:

Advice. Deep copy `can be` implemented for passing function arguments and assignment `can be` redefined to use deep copy. However, it is probably not desirable in most cases because of the great expense of allocating and initializing copies of large data structures. Object-oriented style involves constant function calls. If a deep copy is made of every dynamic object every time a function is called, performance will suffer badly. It is "safe", but the cost of that safety may be too great. A much more efficient way to handle arguments that is just as safe is to pass the argument as a constant reference.

6.4.2 Dangling pointers.

A dangling pointer is a pointer that refers to an object that has been deallocated. Using a dangling pointer is likely to cause amnesia (problem 2. In the following program, a dangling pointer is created when the function `badread()` returns a pointer to a local array. (My compiler did give a warning comment about it.) The string "hijk" was in that array when the function returned, but was freed and overwritten by some other part of the program before the output statement in the last line.

```
27  #include "tools.hpp"
28  char* badread( int n ){
29      char buf[n];
30      cin.get( buf, n );
31      return buf;
32  }
33  //-------------------------------------------------
34  char* goodread( int n ){
35      char* buf = new char[n];
36      cin.get( buf, n );
37      return buf;
38  }
39  //-------------------------------------------------
40  int main(void)
41  {
42      cout << "Enter the alphabet: ";
43      char* word1 = goodread( 8 );
44      char* word2 = badread( 5 );
45      char* word3 = goodread( 8 );
46      cout << "Your letters: " <<word1 <<" " <<word2 <<" "<<word3 <<endl;
47  }
```

Output:

```
Enter the alphabet: abcdefghijklmnopqrstuvwxyz
Your letters: abcdefg  lmnopqr
```

6.4.3 Storing things in limbo.

Limbo is an unknown, random place. Storing things in limbo can cause problem 2, 3, or 4. A common error is to try to store character data in a char* variable that does not point at a character array, as shown in the program below. On my Mac OS-X system, the first input line, `cin >> fname >> lname`, causes a bus error. The compiler does not detect this error because, to it, char* and char[] are compatible types. Caution: there should be a call on `new` or `malloc()` for every char* variable that holds input.

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

```
50  typedef char* cstring;
51
52  int main( void )
53  {
54      char* fname;                  // Not initialized to anything.
55      cstring lname;                // Points off into limbo.
56      cout << "Enter two names: ";
57      cin >> fname >> lname;        // Storing in Limbo..
58      cout << fname << " " << lname << endl;
59  }
```

Output:

```
Enter two names: Andrew Bates

Limbo has exited due to signal 10 (SIGBUS).
```

6.4.4 The wrong ways to delete.

Shallow copy and deep deletion. Class objects with extensions should be passed by reference, not by value. If you pass a class object to a function as a call-by-value argument, a shallow copy of it will be used to initialize the parameter. When the function returns, the class constructor will be executed on the parameter. If the argument had an extension, and the destructor is written in the normal way, the extension will be deleted and the original copy of the object will be broken. Later, when the freed memory is reallocated, the program will experience amnesia (problem 2). This extension of the output operator, and the call on it in the third line, illustrate the most common deep-delete error:

```
inline ostream& operator<<( ostream& out, Dice d ) { return d.print(out); }
Dice mySet(5);
cout << mySet;
```

The second line declares a set of 5 dice, which are implemented by a dynamically allocated array of 5 integers. The Dice destructor (properly) deletes that array. The third line calls the faulty output operator, and, as part of the call, makes a copy of mySet (which shares extensions with the original). When printing is finished, this copy will be freed, its destructor will be run, and the dice array will be gone. This is Not the intended result!

Wrongful deletion. Trying to delete a static or auto object may cause problem 3 or 4 or 5 or may result in a delayed error. If a pointer stores any address other than one that was returned by `new`, it must never be the argument of `delete`. In the next example, we try to delete an auto array. The result is a segmentation fault.

```
57  #include <iostream>
58  using namespace std;
59
60  int main( void )
61  {
62      char buf[80];
63      char* input = buf;
64      cout << "Enter some wisdom: ";
65      cin.getline( input, 80 );
66      cout << "Today's words for the wise: " <<input <<endl;
67      delete input;
68      cout << "Normal termination";;
69  }
```

Output:

```
Enter some wisdom: To forgive is difficult.
Today's words for the wise: To forgive is difficult.
wisdom(43266) malloc: *** error for object 0x7fff5fbff410: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort
```

Partial deletion. The same kinds of disasters will happen when you delete through a pointer that points at the middle of a dynamically allocated array, as in the next example. The newest Gnu C++ compiler was used here; it trapped the faulty delete command and produced an intelligent error comment instead of crashing. Older compilers simply crash with a segmentation fault.

```
69  #include "tools.hpp"
70  int main(void)
71  {
72      char* scan, * buf = new char[80];
73      cout << "Enter some words and some punctuation: \n\t";
74      for(;;) {
75          cin.getline( buf, 80 );
76          scan = strchr( buf, '.' );
77          if (scan != NULL) break;
78          cout << "I don't like " <<buf <<" Try again.\n\t";
79      }
80      cout << "I like the last one.\n" <<scan <<endl;
81      delete scan;
82  }
```

Output:

```
Enter some words and some punctuation:
        It is sunny today!
I don't like It is sunny today! Try again.
        It is rainy today.
I like the last one.
.
delestack(43229) malloc: *** error for object 0x100100091: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort
```

Double deletion. Deleting the same thing twice is likely to cause problem 4, 3 or a delayed fatal error. The outcome would depend on the memory management algorithm used by your compiler. This commonly happens when the program correctly deletes through the pointer to the head of a dynamic area, but also deletes through a copy of the head pointer. This error is trapped by the new Gnu C++ compiler.

Failure to delete. If an area is not deleted when its useful life is over, problem 1 , a memory leak, will result. This is a serious problem for real-life programs that run in the background for long periods of time.

6.4.5 Walking on memory.

```
83  #include "tools.hpp"
84  int main( void )
85  {
86      char fname[10], lname[10];
87      int nums[4], total = 0;
88      cout <<"Enter two names: ";  cin  >>fname >>lname;
89      cout <<"You typed: " <<fname << " " << lname << endl;
90      cout <<"Enter four numbers: ";
91      for( int k=1; k<=4; ++k)  {
92          cin >> nums[k];
93          total += nums[k];
94      }
95      cout <<"First: " <<fname << "\nLast: " <<lname <<"\nTotal = " <<total <<endl;
96  }
```

Output:

```
Enter two names: Margaret Wilkenson
You typed: Margaret Wilkenson
Enter four numbers: 2 3 4 5
First: Margaret
Last:
Total = 14
```

We say that a program "walks on memory" when it stores data beyond the end of the area (usually an array) that has been allocated for the target object. This can cause problem 2, 4, or 3. This happens when:

1. You store beyond the end of the allocated array because of careless array processing. (See example below.)
2. The input string is longer than the buffer during a call on an unlimited string input function.
3. The "from" string is longer than the "to" array during a call on strcpy().
4. The "to" array is too short to contain the combined strings during a call on strcat().

In the next program, the array `nums` is used to hold a series of numeric inputs. The program runs, reads input, and (because of careless use of subscripts) stores the last number past the end of the `nums` array, on top of the array allocated for the last name. The results are machine dependent, but on both of my systems, we get the output shown below the program. Note that the last name was correctly printed on the second line, but it is gone from the fifth line. Why? When the loop went past the end of the nums array, it stored the number 5 (0x05) on top of the last name. The 0 byte (`'\0'`) fell on top of the 'W', so nothing was printed.

Parentheses are not square brackets. Square brackets [ ] are used to allocate an array. Parentheses () are used to enclose an argument for a constructor function or an initializer for a variable. Using () when you need square brackets can cause amnesia (problem 2) because the program will allocate one variable, not an array of variables, and eventually will overrun memory. In the next program, we want to allocate a new array of chars, but we use parentheses where the brackets belong. The line still looks "normal", and the mistake is easy to make and easy to miss:

```
 97   #include "tools.hpp"
 98   int main( void )
 99   {
100       char* buf1 = new char(100);
101       char* buf2 = new char(100);
102       cout <<"Enter two lines of text: \n";
103       cin.getline( buf1, 100 );
104       cin.getline( buf2, 100 );
105       cout <<"Echo [(" <<buf1 <<endl <<buf2 <<")]\n" ;
106   }
```

Output:

```
Enter a line of text:
I once saw a doggie in the window,
Echo [(I once sEcho [(I once sEcho [(I on)]
```

The result is that you THINK there is enough space for a long string, but only 1 byte (initialized to the letter 'd', whose ASCII code is 100), plus padding was provided. In this case, it appears that eight bytes were allocated altogether, but some compilers might allocate less. Your program will probably not bomb, but when you (or the run-time system) allocates the next new object, all but the first N bytes of the data will be overlaid by new data. (Where N-1 is the number of bytes that were used for padding.) The output in this particular case is bizarre: I can't explain why there are three copies of some letters, and this result would probably not be repeatable on another system.

Changing a string literal. Many compilers store literals in a read-only storage area. An attempt to change the contents of a string literal can cause problem 4.

6.4.6 NULL pointers.

To avoid trouble, be careful not to walk off the end of a linked list, process an uninitialized portion of an array, or delete the referent of a `NULL` pointer. An attempt to dereference a NULL pointer may cause problem 4, or 3, depending on your system. The same is true of any attempt to apply a subscript or `++` to the NULL pointer. The nature of the error will be system dependent. I have three computers with different processors and different operating systems, all running the gnu C++ compiler. On one computer, I got a segmentation error, on the other two a bus error, as shown below.

```
105  #include "tools.hpp"
106
107  int main( void )
108  {
109      char* name[5] = {"Jane", "Yonit", "Theo", "Xin"};
110      char* selection;
111
112      for(int k=0; k<5; k++){
113          selection = name[k];
114          cerr << "Name [" << k <<"] is " << selection << " \t";
115          cerr << "[" << k <<"][" << k <<"] = "<<selection[k] << "\n";
116      }
117      bye();
118  }
```

Output:

```
Name [0] is Jane        [0][0] = J
Name [1] is Yonit       [1][1] = o
Name [2] is Theo        [2][2] = e
Name [3] is Xin         [3][3] =
Name [4] is (null)      Bus error
```

Printing. On many systems, garbage will be printed if you attempt to print a string but the string pointer is NULL On my system, however, it is always safe to print a string because the error is trapped by the print routines, and the output printed is `(null)`, as in the output above.

Illegal access. The program on line 119 is a simplified version of the Partial deletion program on line 69. The error that caused the earlier version to crash was eliminated. However, this still crashes with problem 3 or 4 or 5 if the input does not contain the "?" that we are searching for. The problem happens because `strchr()` returns a NULL pointer when it does not find the required character. That is not a problem. However, line 127 uses the result of the `strchr()` without testing it first and sends the memory address 1 to `<<`. This address is used by the system's BIOS (basic input-output system) and must not be used by an application program. On a system with memory protection, the result will always be a fatal error.

```
119  #include <iostream>
120  using namespace std;
121
122  int main(void)
123  {
124      char buf[80];
125      cout << "Enter some words and some punctuation: \n\t";
126      cin.getline( buf, 80 );
127      char* scan = strchr(buf, '?');
128      cout << "I like the last part: " <<scan+1 <<"\n\n";
129      cout << "buf: " <<buf <<endl;
130      cout << "Normal termination";
131  }
```

Output:

```
Enter some words and some punctuation:
        Hungary. Turkey, Chile!
Segmentation fault
```

Garbage in. Garbage in the input stream can cause problem 6. For example, each line in a DOS text file ends in two characters: carriage-return line feed. Unix lines have only one char at the end of a line. If you attempt to read a DOS data file in a program compiled for UNIX, the end-of-line characters will be garbage and will probably cause an infinite loop. The I/O demonstration program at the end of Chapter 3 illustrates this and other file-handling issues.