3 Arrays and Strings

Beyond the four basic data types (char, int, float, double), the C language supports some advanced data constructs. These include arrays, strings, pointers and structures. Like the basic data types, each of these advanced data constructs is intended to store something different:

Data construct	Intended to store:
array	list of same-type values
string	text
pointer	address of another variable
structure	group of mixed-type values

An array is intended to group together a list of values, all of the same type, under one variable name. It is much easier to code computations on an array than on a list of independently named variables:

```
for (i=0; i<100; i++)      /* array coding */
  sum=sum+qty[i];

sum=qty1+qty2+qty3+...     /* without an array */
```

A structure is intended to group together an assortment of values of different types under a single variable name. Like an array, it makes it easier to code computations. A string is intended to group together a series of character symbols under one variable name. It is closely related to an array, but because it is intended only for text data, a number of functions have been crafted to perform text-specific operations on strings. A pointer is intended to hold the address of another variable, to provide a "gateway" or path of *indirect access* to the other variable. It is used most often in passing values between pieces of code (functions). This chapter examines arrays and strings; pointers and structures are discussed in the next chapter.

It is assumed that the reader is familiar with the basic syntax and use of arrays and strings (although this will be briefly reviewed). The first purpose of this chapter is to describe *how* arrays and strings work, by examining how they are constructed in memory. To accomplish this, we will use the memory map. An understanding of how each data construct resides in memory is useful for program design, and especially debugging. The second purpose of this chapter is to describe the functions provided by the C standard library that operate on string data. Once familiar with the concepts of strings, a programmer saves time and effort by using this system resource.

3.1 • Arrays

An array is a construct used to store a set of values using only one variable name. Each of the values occupies a *cell*. Every cell in the array is the same size, meaning it occupies the same amount of memory. The size of each cell is dictated by the data type (char, int, float, double) given in the variable declaration. The number of cells is also given in the variable declaration. Here are some examples:

```
int a[2];      /* 2 cells, each cell 4 bytes (32 bits) */
float b[3];    /* 3 cells, each cell 4 bytes (32 bits) */
double c[4];   /* 4 cells, each cell 8 bytes (64 bits) */
char d[5];     /* 5 cells, each cell 1 bytes (8 bits) */
```

Cells are accessed using *indices*, with syntax similar to that used in the variable declarations. For example:

```
a[0]=5;
b[1]=4.0;
c[2]=14.7;
d[4]='a';
```

The memory map for these code segments could be written as:

Label	Address	Value
a[0]	400–403	5
a[1]	404–407	
b[0]	408–411	
b[1]	412–415	4.0
b[2]	416–419	
c[0]	420–427	
c[1]	428–435	
c[2]	436–443	14.7
c[3]	444–451	
d[0]	452	
d[1]	453	
d[2]	454	
d[3]	455	
d[4]	456	'a'

Most of the cells in the arrays were not given values in this code example. Therefore their values are unknown. It is convenient to leave these entries blank in the memory map table. This does not mean that they do not have a value, just that their values are unknown (see Chapter 2).

When accessing an array, the index used should be from zero to one less than the total number of cells in the array. These are called the *array bounds*. However, the C language does not check to make sure that program code stays within the array bounds. It is possible to compile and execute code that goes outside the array bounds. For example, if we add the following code to that from above:

```
b[4]=15.9;
printf("%lf\n",b[4]);
```

Even though index 4 is past the array bounds for the variable b, the output of this code still produces the expected output:

```
15.900000
```

Why did it work? Looking at the memory map, one can see what happens. The four bytes after b[2] can be accessed using the name b[3]:

Out-of-bounds access	Label	Address	Value
	a[0]	400–403	5
	a[1]	404–407	
	b[0]	408–411	
	b[1]	412–415	4.0
	b[2]	416–419	
b[3] ? b[4] ?	c[0]	420–427	
b[5] ? b[6] ?	c[1]	428–435	
	c[2]	436–443	14.7
	c[3]	444–451	
	d[0]	452	
	d[1]	453	
	d[2]	454	
	d[3]	455	
	d[4]	456	'a'

In the memory map, these bytes occupy addresses 420–423 and are used by the variable c[0]. However, the C compiler allows a programmer to access them by "going off the end" of the b array. Past the array bounds, accesses simply move ahead the appropriate number of bytes. The next four bytes, at memory addresses 424–427, can be accessed using the name b[4]. These operations "work" because although they are accessing memory outside the array bounds, they are still accessing memory used by this program. Therefore, the program compiles and executes with seemingly correct output.

Of course, it is possible for an access outside the array bounds to *clobber* a variable. This happens when the out-of-bounds array access overwrites a value in another variable. In our example above, half the variable c[0] is clobbered by writing to b[3], and the other half of c[0] is clobbered by writing to b[4]. Encountering this error, a programmer will observe a variable to suddenly change value. Seemingly, no line of code in the program changes that variable's value, yet when displayed it has changed. This mistake often frustrates novice programmers. With some practice, one will recognize that it is likely the result of an out-of-bounds array access that has clobbered another variable.

It is also possible for an out-of-bounds array access to cause a program to crash. For example, consider adding the following code fragment to those from above:

```
b[33333]=15.9;
printf("%lf\n",b[33333]);
```

This code still compiles, but when executed produces a segmentation fault. This is because the out-of-bounds array access went far beyond the memory used by this program, and perhaps tried to access memory used by another program (or that was otherwise restricted). The operating system will recognize the problem and terminate the program, reporting a segmentation fault.

Faulty addressing can lead to another type of crash called a *bus error.* A bus error occurs when a program tries to access a memory address that is physically impossible (or nonexistant). A bus error can also occur on some systems when a program tries to address an "unaligned" address. This means that the program attempts to read multiple bytes that are not aligned with the width of the data bus (for example, on a 32-bit system the data bus is 4 bytes wide). If an error of this type happens, the operating system will recognize the problem and terminate the program, reporting a bus error.

Why does the C compiler allow out-of-bounds array accesses? The question is actually more complicated than it seems. Not all arrays have a fixed, known size. Sometimes a programmer needs to decide on the size of an array after a program has been compiled and is executing. Sometimes a programmer needs an array to change size while a program is executing. These needs make it impossible for the C compiler to know, in all cases, if an array access is out-of-bounds. The answer may not be known until the program is running and reaches the code that accesses the array.

3.1.1 Multidimensional Arrays

Arrays in C can have more than one dimension. But computer memory is all arranged in one-dimensional order, as though it were one long street of bytes. How then are multidimensional arrays stored in computer memory? Does a computing system have some other strange, multidimensional space for use only by these arrays? Of course not. The answer is that the cells in the multidimensional array are listed out, one at a time, in one-dimensional order. For example, consider the following code:

```
int a[3][2];
a[0][1]=7;
a[1][0]=13;
```

The memory map for this code could be written as:

Label	Address	Value
a[0][0]	400–403	
a[0][1]	404–407	7
a[1][0]	408–411	13
a[1][1]	412–415	
a[2][0]	416–419	
a[2][1]	420–423	

There are a total of six cells in this array. They are listed in order by cycling through the range on the rightmost index, incrementing the next index to the left when done. The same procedure works for any number of dimensions. For example, consider the following code for a three-dimensional array:

```
int b[2][3][4];
b[0][2][0]=7;
b[1][0][2]=13;
```

There are 24 cells in this array, occupying a total of 96 bytes. The memory map for this code could be written as:

Label	Address	Value	Label	Address	Value
b[0][0][0]	400–403		b[1][0][0]	448–451	
b[0][0][1]	404–407		b[1][0][1]	452–455	
b[0][0][2]	408–411		b[1][0][2]	456–459	13
b[0][0][3]	412–415		b[1][0][3]	460–463	
b[0][1][0]	416–419		b[1][1][0]	464–467	
b[0][1][1]	420–423		b[1][1][1]	468–471	
b[0][1][2]	424–427		b[1][1][2]	472–475	
b[0][1][3]	428–431		b[1][1][3]	476–479	
b[0][2][0]	432–435	7	b[1][2][0]	480–483	
b[0][2][1]	436–439		b[1][2][1]	484–487	
b[0][2][2]	440–443		b[1][2][2]	488–491	
b[0][2][3]	444–447		b[1][2][3]	492–495	

The order of incrementing dimension indices is similar to how base 10 numbers are counted, cycling through the one's digit before incrementing the ten's digit, then cycling through the ten's digit before incrementing the hundred's digit, and so on.

Everything stored in computer memory is somehow stretched out in one-dimensional order. Things not typically viewed as one-dimensional, such as images, video, databases, maps, and three-dimensional models are all actually stored by listing out the data in one-dimensional order. We will revisit this principle in Section 5.4 when we look at files.

3.2 • Strings

A string is a specific type of array: it is an array of char, containing a sequence of values where a value of `'\0'` signifies the end of the string. Although the array could be of any size, it is assumed that the valid data in the array starts at the first cell, and ends with the first cell having a value of `'\0'`. For example:

```
char d[8];
d[0]='H'; d[1]='e'; d[2]='l'; d[3]='l'; d[4]='o';
d[5]='\0';  /* '\0' indicates the end of string */
```

The memory map for this code could be written as:

Label	Address	Value
`d[0]`	400	`'H'`
`d[1]`	401	`'e'`
`d[2]`	402	`'l'`
`d[3]`	403	`'l'`
`d[4]`	404	`'o'`
`d[5]`	405	`'\0'`
`d[6]`	406	
`d[7]`	407	

Although the array has eight cells, only the first six are used. The cell containing the `'\0'` character is not seen during either printing or the scanning of input. It is a *nonprintable* character[1] used to control how text data is processed. Any code working on a string is supposed to stop processing the array when the `'\0'`

1. The slash-symbol pair within the single quotes is called an *escape sequence* or *control sequence* and is used to represent the nonprintable character because there is no single visible symbol to represent the action.

character is reached. It is assumed that the values in the remaining cells (two in this case) are not used.

The value of the `'\0'` character is zero. `'\0'` is simply another way of saying zero. It is used to specify a specific "type" of zero. For example, it is informative to use a different zero for a whole number (0) than for a real number (0.0) to show that they are representing slightly different information. `'\0'` is a way to say zero for a char representing an ASCII symbol that means end of string. Zero has another alias called NULL. NULL is usually used to indicate a value of zero for an address (discussed further in Chapter 4), but it is sometimes used for the end of string character. Any of these aliases can be used; they all result in the same bit pattern being placed in the cell, which is all zeros. For example, all of the following lines of code are equally valid and do the exact same thing:

```
d[5]='\0';        /* ASCII zero */
d[5]=0;           /* integer zero */
d[5]=(char)NULL; /* address zero */
```

The basic functions for reading string input from the keyboard and for writing string output to the screen are `scanf()` and `printf()`, respectively. Just as each basic data type has its own % identifier (char uses %c, int uses %i or %d, float uses %f, double uses %lf), a string uses %s. The %s identifier makes it simpler for a programmer to print out a string. How does it work? Consider the following code to print out the example from above:

```
printf("%c%c%c%c%c\n",d[0],d[1],d[2],d[3],d[4]);
```

This brute force approach is tedious; code must be written to specifically print out each character, and we must know exactly how many are to be printed. The %s identifier tells printf() to assume the variable is an array of char, ending with a value of zero, and to print each byte using the ASCII bit model. Using this identifier, we can rewrite the printf() as follows:

```
printf("%s\n",d);
```

Notice that there are no indices given for the variable `d` in the printf() when using the %s identifier. The printf() is given the variable name `d`, not `d[0]` through `d[4]`. We can explain this using the memory map:

Address label	Label	Address	Value
d	d[0]	400	'H'
	d[1]	401	'e'
	d[2]	402	'l'
	d[3]	403	'l'

	d[4]	404	'o'
	d[5]	405	0
	d[6]	406	
	d[7]	407	

The variable name d is a label for an address rather than for a value. It identifies address 400. Given that address, the printf() function will print out bytes until it encounters a value of zero. Thus, it will print out the bytes at addresses 400–404, and upon seeing a value of zero at address 405, it stops. What if we were to code the following?

```
printf("%s\n",d[0]);  /* what does this do? */
```

This code does not make any sense. It seemingly asks printf() to print character symbols, starting with the value 'H'. But which 'H'? How is printf() supposed to know that the programmer intended printing to start with the specific 'H' at address 400? In fact, this code will crash. It asks printf() to start printing bytes at "address 'H'", which causes a segmentation fault.

When reading input using the scanf() function, the & symbol is required in front of the variable name for the basic data types (char, int, float, and double). However, it is not required for a string. For example:

```
int x;
float f;
char s[6];
scanf("%d",&x);
scanf("%f",&f);
scanf("%s",s);
```

This can be explained by looking at the memory map:

Address label(s)	Label	Address	Value
&x	x	400–403	
&f	f	404–407	
&(s[0]) s	s[0]	408	
	s[1]	409	
	s[2]	410	
	s[3]	411	
	s[4]	412	
	s[5]	413	

The variable names x and f each refer to the values stored for those variables. Similarly, s[0] through s[7] refer to values. The syntax &x is used to identify the address of x, which is 400. The syntax &f identifies the address of f, which is 404, and &(s[0]) identifies the address of s[0], which is 408. The variable name s is simply a shorthand for writing &(s[0]). The nature of the operation involving the %s identifier in scanf() is to store a series of characters, ending it with a value of zero. This is why it must be given an address, where the storing of characters is to start.

3.2.1 Multidimensional Strings

One of the common uses for multidimensional arrays is to store a list of strings. Since each string is a one-dimensional array, a list of strings requires a two-dimensional array. For example:

```
char n[2][4];
n[0][0]='T'; n[0][1]='o'; n[0][2]='m'; n[0][3]=0;
n[1][0]='S'; n[1][1]='u'; n[1][2]='e'; n[1][3]=0;
```

The memory map for this code can be written as:

Address label(s)	Label	Address	Value
&(n[0][0]) n[0] n	n[0][0]	400	'T'
	n[0][1]	401	'o'
	n[0][2]	402	'm'
	n[0][3]	403	0
&(n[1][0]) n[1]	n[1][0]	404	'S'
	n[1][1]	405	'u'
	n[1][2]	406	'e'
	n[1][3]	407	0

Each string in the two-dimensional array can be referenced using the address label for the one-dimensional array that stores it. For example, the following code prints out the two strings:

```
printf("%s %s\n",n[0],n[1]);
```

Both n[0] and n[1] identify addresses (400 and 404, respectively). In the case of a multidimensional array, the variable name by itself provides a third alias for the starting address of the entire array. In this example, &(n[0][0]), n[0], and n all refer to the same thing, the address 400.

3.3 • String Library Functions

There are a handful of calculations that are common to a large number of text processing problems. These calculations include finding the length of a string and comparing the contents of two strings. Because they are so common, the C standard library has evolved to include functions to perform these calculations. In Chapter 8 (Section 8.3) we take a deeper look at the C standard library, which contains functions for a variety of purposes; in this section, we are concerned only with the portion of the library involved in text processing. Even that portion of the library includes dozens of functions.[2] However, there are five functions that cover the most common calcuations and operations:

Function	What it does
strlen()	count the total characters in the string
strcmp()	compare two strings, determine if identical
strcpy()	copy one string into another string variable
strcat()	append one string to another string
sprintf()	print formatted output into a string variable

The following sections examine each of these functions from the perspective of memory. Studying the operations at the memory level is a good way to approach many text processing problems and is particularly helpful when tackling problems outside the scope of the string library functions.

3.3.1 String Length: strlen()

Suppose we wish to count the number of characters in a string. For example:

String	Length
"Hello"	5
"H.i;"	4
"h e y"	5

We can calculate the length of a string using the following code:

2. For a complete listing, consult a reference such as *C: A Reference Manual*, 5th ed., S. P. Harbison and G. L. Steele, Prentice Hall, 2002.

```
int length;
char s[6];
s[0]='S'; s[1]='u'; s[2]='e'; s[3]='\0';
length=0;
while (s[length] != '\0')
  length++;
```

The memory map during execution of this code may be written as:

Label	Address	Value
`length`	400–403	~~0~~ ~~1~~ ~~2~~ 3
`s[0]`	404	`'S'`
`s[1]`	405	`'u'`
`s[2]`	406	`'e'`
`s[3]`	407	`'\0'`
`s[4]`	408	
`s[5]`	409	

Notice how the `'\0'` value is used to control the calculation. Upon reaching it, the calculation ends. This type of value is sometimes called a *sentinel* or a flag. The value of a sentinel is not intended to be used in a computation; rather, it is intended to terminate the processing of a list of data. Sentinels are used in many computational problems besides text processing.

The same calculation can be performed by calling the `strlen()` function:

```
length=strlen(s);
```

It does not save us a lot of code for this one calculation (1 line versus 3 lines of code). However, after having written code for the same calculation a few hundred (or thousand) times, the savings add up. In addition, calling the same function prevents accidentally inserting an error into the calculation. Even experienced programmers can make mistakes, and the benefit of using proven code can save the time it takes to track down bugs.

3.3.2 String Compare: strcmp()

Suppose we wish to compare two strings to determine if they are the same. Further, if they are different, we desire to know which string first reaches an index having a value less then the same index in the other string. For example:

Strings	Comparison (meaning)
“Hello” vs. “Hello”	0 (same)
“Hello” vs. “Hellp”	−1 (first string smaller)
“Hey” vs. “Hallo”	1 (second sting smaller)
“Hillo” vs. “Hi”	1 (second sting smaller)

We can compare two strings using the following code:

```
int i,a;
char s[4],t[4];
s[0]='S'; s[1]='u'; s[2]='e'; s[3]='\0';
t[0]='S'; t[1]='u'; t[2]='n'; t[3]='\0';

i=0; a=0;
while (a == 0)
  {
  if (s[i] < t[i]) a=-1;
  if (s[i] > t[i]) a=1;
  if (s[i] == '\0'  ||  t[i] == '\0')
    break;
  i++;
  }
```

The memory map during execution of this code may be written as:

Label	Address	Value
i	400–403	~~0~~ ~~1~~ ~~2~~ 3
a	404–407	~~0~~ -1
s[0]	408	'S'
s[1]	409	'u'
s[2]	410	'e'
s[3]	411	'\0'
t[0]	412	'S'
t[1]	413	'u'
t[2]	414	'n'
t[3]	415	'\0'

The loop compares the values in the cells at the same indices of the two strings (s[0] to t[0], s[1] to t[1], etc.) until it finds an index where the values are different, or until it reaches the end of one of the strings. The comparison result is stored in a. If the strings are the same, the comparison result is 0. If the first string is less than the second, the result is −1; if the first string is greater than the second, the result is 1. This provides an alphabetical comparison, so long as the case of the letters is equivalent. For our example, since "Sue" is less than "Sun," the comparison result is −1.

The same calculation can be performed by calling the `strcmp()` function:

```
a=strcmp(s,t);
```

This saves us slightly more code than the strlen() function, but like strlen(), its real value lies in repeated use.

3.3.3 String Copy: strcpy()

Suppose we wish to copy the contents of a string to a second string variable. The following code accomplishes this task:

```
int i;
char s[4],t[4];
s[0]='S'; s[1]='u'; s[2]='e'; s[3]='\0';
i=0;
while (s[i] != '\0')
  {
  t[i]=s[i];
  i++;
  }
t[i]='\0';
```

The memory map during execution of this code may be written as:

Label	Address	Value
i	400-403	~~0~~ ~~1~~ ~~2~~ 3
s[0]	404	'S'
s[1]	405	'u'
s[2]	406	'e'
s[3]	407	'\0'
t[0]	408	'S'
t[1]	409	'u'

t[2]	410	'e'
t[3]	411	'\0'

Notice that the loop will not copy the '\0' character. It must be added to the new string separately after the loop is completed. This is a common programming mistake. Without the '\0' character, any processing of the new string will be erroneous. Any code or function processing that string will continue until it finds a zero byte somewhere in memory (which could cause a crash or other problem).

The same calculation can be performed by calling the `strcpy()` function:

```
strcpy(t,s);
```

The source string comes second in the argument list; the destination string comes first.

3.3.4 String Concatenate: strcat()

Suppose we wish to append a string (the addendum) to the end of another string (the original). For example:

Original	Addendum	Result
"Hi"	" there"	"Hi there"
"Sun"	"ny"	"Sunny"
"a"	".out"	"a.out"

We can concatenate two strings using the following code:

```
int i,j;
char s[8],t[4];
s[0]='S'; s[1]='u'; s[2]='\0';
t[0]='s'; t[1]='a'; t[2]='n'; t[3]='\0';
i=strlen(s);
j=0;
while (t[j] != '\0')
  {
  s[i+j]=t[j];
  j++;
  }
s[i+j]='\0';
```

The memory map during execution of this code may be written as:

Label	Address	Value
`i`	400–403	2
`j`	404–407	~~0~~ ~~1~~ ~~2~~ 3
`s[0]`	408	`'S'`
`s[1]`	409	`'u'`
`s[2]`	410	~~0~~ `'s'`
`s[3]`	411	`'a'`
`s[4]`	412	`'n'`
`s[5]`	413	0
`s[6]`	414	
`s[7]`	415	
`t[0]`	416	`'s'`
`t[1]`	417	`'a'`
`t[2]`	418	`'n'`
`t[3]`	419	0

In this example, we make use of another string library function (`strlen()`) to accomplish the task. It is common to see library functions that make use of other functions (or even other libraries). We will look at this principle in Chapter 8 when we examine libraries in general. In this example, we also had to explicitly add the `'\0'` character at the end of the result so that the string remains valid for further processing.

The same calculation can be performed by calling the `strcat()` function:

```
strcat(s,t);
```

The original string comes first in the argument list; the addendum string comes second. The result is placed in the original string variable.

3.3.5 String Print: sprintf()

The `sprintf()` function works just like the `printf()` function, except that the output "prints" into a string variable. This can be useful for converting numeric data types into ASCII text, or for creating long strings from multiple components. For example:

```
char a[24];
float f;
int i;
```

```
f=3.72;
i=9;
sprintf(a,"Price %f, qty %d",f,i);
printf("%s\n",a);
```

The output of this code is:

```
Price 3.720000, qty 9
```

The memory map for this code is revealing; it highlights the difference between text-storage and value-storage of numbers:

Label	Address	Value	Label	Address	Value
a[0]	400	'P'	a[13]	413	'0'
a[1]	401	'r'	a[14]	414	','
a[2]	402	'i'	a[15]	415	' '
a[3]	403	'c'	a[16]	416	'q'
a[4]	404	'e'	a[17]	417	't'
a[5]	405	' '	a[18]	418	'y'
a[6]	406	'3'	a[19]	419	' '
a[7]	407	'.'	a[20]	420	'9'
a[8]	408	'7'	a[21]	421	0
a[9]	409	'2'	a[22]	422	
a[10]	410	'0'	a[23]	423	
a[11]	411	'0'	f	424–427	3.72
a[12]	412	'0'	i	428–431	9

How many bytes does it take to store each of the numbers as a numeric value (float or int) versus as text?

3.3.6 String Functions Example

The following code demonstrates several of the string functions just covered, together in a complete program:

```
#include <stdio.h>     /* for printf(), scanf() */
#include <string.h>    /* for strlen(), strcmp() */
main()
{
char    look[80],test[80];
printf("Look for: ");
scanf("%s",look);
while (1)
  {
  printf("Enter a string (0 to quit): ");
  scanf("%s",test);
  if (strcmp(test,"0") == 0)
    break;
  if (strlen(test) < strlen(look))
    printf("%s is too short for %s\n",test,look);
  else if (strcmp(test,look) == 0)
    printf("Found one!\n");
  else if (strncmp(test,look,3) == 0)
    printf("Started the same...\n");
  else
    printf("Not what we're looking for\n");
  }
}
```

Compiling and executing the code, we obtain the following (the text given at each input was selected to demonstrate the various functions):

```
Look for: sun
Enter a string (0 to quit): s
s is too short for sun
Enter a string (0 to quit): sun
Found one!
Enter a string (0 to quit): sunny
Started the same...
Enter a string (0 to quit): sleet
Not what we're looking for
Enter a string (0 to quit): 0
```

The last function call, `strncmp()`, is a variant on `strcmp()`. It takes a third argument indicating how many characters (three in this example) are to be compared. If no difference is found between the two strings after that many characters have been compared, then the function returns zero. Otherwise, the strncmp() function works exactly the same as the original strcmp() function. There are similar variants on the `strcpy()` and `strcat()` functions, called `strncpy()` and `strncat()`.

3.3.7 Nonlibrary Problems

A naive programmer can become too reliant upon the string library functions. Not every text processing problem is easily solved through use of the library functions. For many problems, individual character processing of string data must be coded. This section demonstrates an example.

Suppose we want to remove all occurrences of the letter `'a'` from a string, compressing characters to fill any created gaps. For example:

Original	Result
"Saturday"	"Sturdy"
"a ball"	" bll"

There is no single function within the C standard library that will accomplish this task. Instead, we must write code to process the string at the character level. The following code accomplishes `'a'`-removal:

```
int i,j;
char s[6];
s[0]='a'; s[1]='b'; s[2]='a'; s[3]='c'; s[4]=0;
i=0; j=0;
while (s[i] != 0)
  {
  if (s[i] != 'a')
    {
    s[j]=s[i];
    j++;
    }
  i++;
  }
s[j]=0;
```

The memory map during execution of this code may be written as:

Label	Address	Value
`i`	400–403	~~0~~ ~~1~~ ~~2~~ ~~3~~ 4
`j`	404–407	~~0~~ ~~1~~ 2
`s[0]`	408	'~~a~~' 'b'
`s[1]`	409	'~~b~~' 'c'

(continued)

Label	Address	Value
s[2]	410	'~~a~~' 0
s[3]	411	'c'
s[4]	412	0
s[5]	413	

A serious programmer should be well-versed in the common string processing functions of the C standard library and should use them when appropriate. However, any serious programmer must also be able to process strings at the character level, in order to accomplish the given task. Studying the problem at the memory level is often a good approach. Writing out one or two examples in memory, especially those involving the trickiest cases, will often facilitate code design and debugging.

3.4 • Command Line Arguments

As discussed in Chapter 1, a command line argument is anything typed at the shell prompt after the name of the program to execute. Command line arguments are typically used to provide information about how the user wishes to run a program. For example:

```
ahoover@video> ls -l -t
drwxr-xr-x  26 ahoover  fusion    4096 Jul 20 16:03 ece222/
drwxr-xr-x   7 ahoover  325       4096 Jul 19 17:01 public_html/
drwx------   2 ahoover  fusion    4096 Feb  7 14:33 mail/
drwxr-xr-x   3 ahoover  fusion    4096 Dec 14  2005 ece854/
drwxr-xr-x   3 ahoover  fusion    4096 Dec 14  2005 ece429/
drwxr-xr-x   2 ahoover  fusion    4096 Mar 19  2005 ece893/
drwxr-xr-x   3 ahoover  fusion    4096 Mar 19  2005 ece468/
drwxr-xr-x  15 ahoover  fusion    4096 Dec 14  2005 Projects/
ahoover@video>
```

The command line argument `-l` tells the program `ls` to provide a long listing in its output. The command line argument `-t` causes the output to be sorted according to time last modified. This example shows two command line arguments; it is possible to have any number. They must be separated from each other by one or more spaces.

How is a program made aware of its command line arguments? When executed, into what variables are the values for the command line arguments placed? The answer lies in the full function declaration for `main()`:

```
int main(int argc, char *argv[])
```

The variable `argc` stores the number of command line arguments, including the name of the program. For the example `ls -l -t`, this equals 3. The variable argv stores a list of strings, one string per command line argument. Although the declaration for argv looks strange, including a pointer symbol and an empty pair of array brackets, it can be accessed just like a two-dimensional array. For example, the following code prints out all the command line arguments, one character at a time:

```
int i,j;
for (i=0; i<argc; i++)
  {
  j=0;
  while (argv[i][j] != '\0')
    {
    printf("%c",argv[i][j]);
    j++;
    }
  printf("\n");
  }
```

The partial memory map for argc and argv, based on our example, may be written as follows:

Address label(s)	Label	Address	Value
&(argc)	argc	400–403	3
&(argv[0][0]) argv[0]	argv[0][0]	404	'l'
&(argv[0][1])	argv[0][1]	405	's'
&(argv[0][2])	argv[0][2]	406	'\0'
&(argv[1][0]) argv[1]	argv[1][0]	407	'-'
&(argv[1][1])	argv[1][1]	408	'l'
&(argv[1][2])	argv[1][2]	409	'\0'
&(argv[2][0]) argv[2]	argv[2][0]	410	'-'
&(argv[2][1])	argv[2][1]	411	't'
&(argv[2][2])	argv[2][2]	412	'\0'

Each string can be accessed through its own address label, so that the code example just given can be simplified as follows:

```
for (i=0; i<argc; i++)
  printf("%s\n",argv[i]);
```

Although the variable argv can be accessed as though it were a two-dimensional array, it actually occupies memory a little differently. Using pointers, what we have been calling the "address labels" in this chapter are given their own storage locations. This is a primary subject of the next chapter.

Questions and Exercises