8. Character Strings

Figure 8.1: Some very early text: The Epic of Gilgamesh, first written down around 2,000 BCE. It tells the story of King Gilgamesh and his friend Enkidu and their epic journey to visit the wise man Utnapishtim, who was a survivor of the Deluge. New fragments of the Epic were discovered in an Iraqi museum in 2015.

Source: Wikimedia Commons

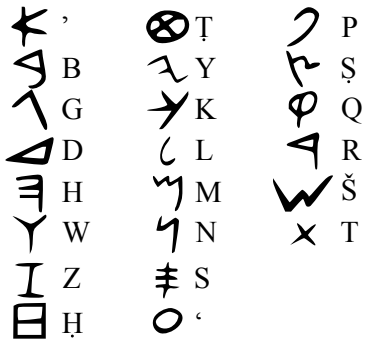


Figure 8.2: The Phoenician alphabet.

Source: Wikimedia Commons

8.1. Introduction

Until now we've avoided reading or writing text in our programs, and have worked exclusively with numbers. Even though we use a lot of numbers in science and engineering, we still need to work with words sometimes. Wouldn't it be convenient to have a header at the top of each column of numbers in a data file, saying what that column means? We might also want to use text for data values themselves, like "`on`" or "`off`" instead of zero or one, to make our data easier for humans to read. Even if we have a glossary of numerical values and their meanings, like "32 = Virginia, 33 = Maryland", it's handy to be able to just look at the data in a file and see its meaning directly, without having to go look up the meaning of each number.

Early writing systems used written symbols to represent the sounds of speech. Learning to read requires that you learn a sort of glossary of these symbols and their speech equivalents. Computers can only store data in the form of binary numbers, so somewhere there's going to need to be a glossary that matches up text with numerical equivalents.

In this chapter we're going to see how computers store text, and how to read, write and compare text in a C program. Although you might not expect it, introducing text also introduces a lot of potential problems for the programmer.

8.2. Character Variables

As we've seen, there are several different types of variables in C. We've used "`int`" for integers and "`double`" for floating-point numbers. Now we're going to introduce another type of variable: "`char`". A `char` variable can hold one character (letter, number, punctuation, etc.)

Here's a C statement that defines a `char` variable named `letter` and gives it the initial value `'A'`:

```
char letter = 'A';
```

Notice that we use single-quotes (apostrophes) around the letter. This tells the computer that `A` isn't the name of a variable, it's literally just the letter `A`. Program 8.1 shows how you might use `char` variables.

Program 8.1: checkyn.cpp

```
#include <stdio.h>
int main () {
  char answer;

  printf ("Can you ride a bike? (y or n): " );
  scanf ("%c", &answer);

  if ( answer == 'y' ) {
    printf ("Yay! Biking is fun.\n");
  } else if ( answer == 'n' ) {
    printf ("Awww. You should learn.\n");
  } else {
    printf ("Might ride a bike, but can't follow instructions.\n");
  }
}
```

Along with the new variable type, we need a new type of placeholder for our printf and scanf statement. Just as we use "`%d`" for `int` and "`%lf`" for `double`, we use "`%c`" for `char`. When we say "`%c`" we mean "insert a single character here".

8.3. Character Strings

We can use an array of `char` elements to hold a chunk of text. We call such an array a "character string" (see Figure 8.4). We'll use the terms "character array", "character string", and "string" interchangeably.

As we saw in Chapter 6, C lets us put numbers into an array when we define it (although this is only practical for small arrays). For example, we could define a small array of integers and print them out like this:

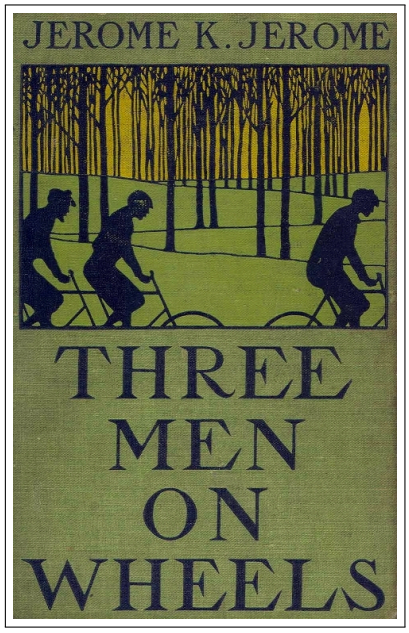


Figure 8.3: *Three Men on Wheels* (1900, aka *Three Men on the Bummel*) by Jerome K. Jerome is a sequel to *Three Men in a Boat (to Say Nothing of the Dog)* (1889). It follows Jerome, George, and Carl on a bicycle trip through Germany.

```
int array[5] = {1,2,3,4,5};
int i;
for ( i=0; i<5; i++ ) {
  printf ( "%d\n", array[i] );
}
```

We could do something similar with an array of characters if we wanted to:

```
char string[20] = {'t','h','i','s',' ',
                   'i','s',' ','a',' ',
                   't','e','s','t','.'};
int i;
for ( i=0; i<20; i++ ) {
  printf ( "%c", string[i] );
}
```

Since we've omitted the `\n` all of the characters will be printed on the same line, and the output will say "`this is a test.`" That's a really tedious way to define a chunk of text and print it out. Fortunately, C provides with a couple of shortcuts to make it easier.

Figure 8.4: Think of a character string as being like a string of letter beads.

First of all, there's a special way of setting the inital value of character strings. Instead of using curly brackets and a list of single-quoted characters, we can just enclose the text in double-quotes:

```
char motto[10] = "Science!";
```

Second, there's a special placeholder, "`%s`", for printing character strings all at once, instead of one character at a time:

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

Notice that we don't have to use all of the elements of a character array. In the example above, the text "Science!" is only eight characters long, but we've defined `motto` to have ten elements. In fact, if we don't plan on ever putting more text into a character string, we can ask the compiler to figure out its length automatically, by just leaving the length blank:

```
char motto[] = "Science!";
```

Of Course, we'll run into trouble if we try to stuff more characters into a character array than it will hold. This would create the same problems we saw in Chapter 6 with other kinds of arrays.

In the following we're going to look at several tiny programs that illustrate some of the problems you might run into when you use character strings in your programs. In each case, we'll show you the "right" way to do it

8.4. How Strings Are Stored

Prior to the 1960s, the most widespread way of communicating data electronically was morse code (see Figure 8.5). When a telegram was sent, its text was encoded in morse code and transmitted through air or a wire to its destination, where it was decoded back into text.

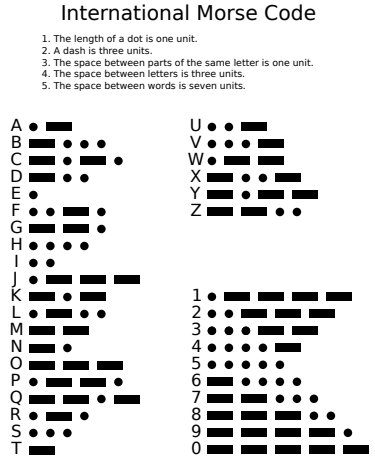


Figure 8.5: Morse Code replaces letters with patterns of dots and dashes.
Source: Wikimedia Commons

Morse code was fine for human telegraphers, but it was clumsy for computers. In the 1960s the "American Standards Association" published a new, more computer-friendly way of transmitting text. This was called the American Standard Code for Information Interchange (ASCII).

In ASCII, each character is represented by 8 bits of information (1 byte). When you store text in a file on disk, the text is stored as ASCII characters. (Actually, other encodings like UTF-8 may be used these days because they allow multi-national characters, but the principle is the same. For simplicity, let's just assume everything is ASCII.)

American Standard Code for Information Interchange (ASCII)

```
01000001 A      01010101 U
01000010 B      01010110 V
01000011 C      01010111 W
01000100 D      01011000 X
01000101 E      01011001 Y
01000110 F      01011010 Z
01000111 G
01001000 H      00110000 0
01001001 I      00110001 1
01001010 J      00110010 2
01001011 K      00110011 3
01001100 L      00110100 4
01001101 M      00110101 5
01001110 N      00110110 6
01001111 O      00110111 7
01010000 P      00111000 8
01010001 Q      00111001 9
01010010 R
01010011 S       ...etc.
01010100 T
```

`00000000 = "NUL"`

Figure 8.6: ASCII code replaces letters with zeros and ones.

8.5. The Length of Strings

Take a look at Figure 8.7, where we define a 10-element character array called `name` and put the word "Fred" into it. If we wanted to print the text stored in `name` we might write a C statement like this:

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

That looks straightforward enough, but it leads to a puzzle: The character array `name` has ten elements, but we're only using four of them. How does the `printf` function know when it gets to the end of the text? In fact, as we noted in Chapter 6, C doesn't prevent us from

reading or writing past the end of an array. Shouldn't we have to tell printf how many characters are in our text, or at least tell it how many elements are in the `name` array? With other types of array, we haven't been able to just say "print the array", so why are we able to do so with character arrays?

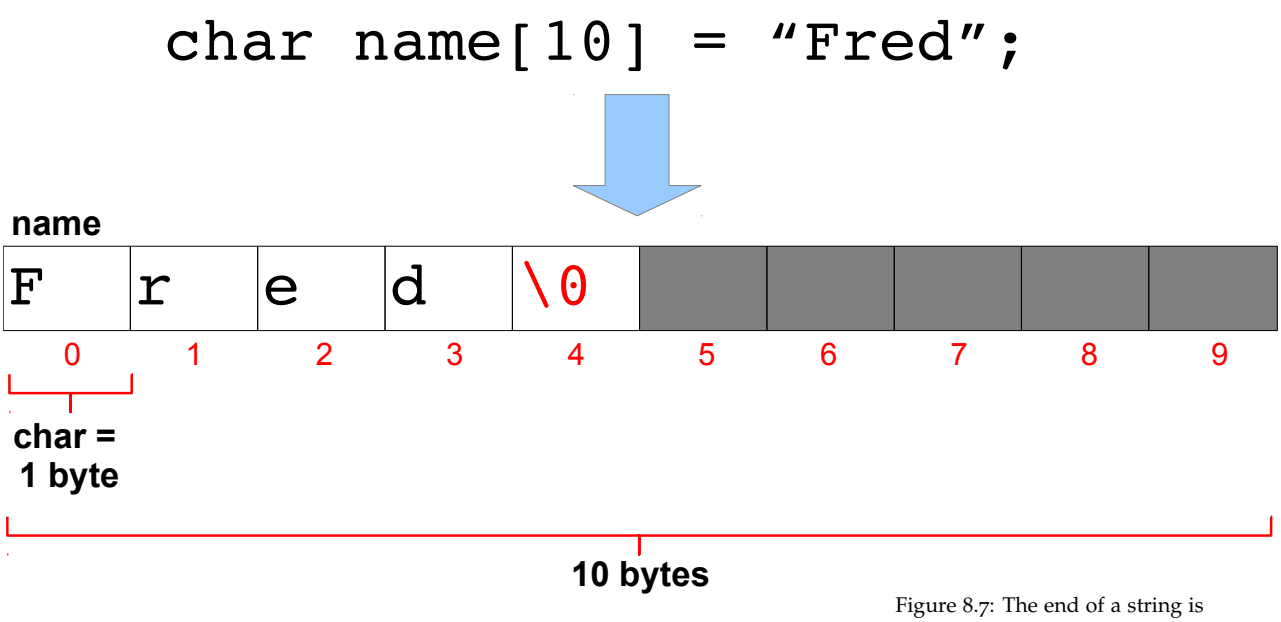


Figure 8.7: The end of a string is indicated by a special non-printable character, the "NUL" character, which we represent by '\0' here (see Figure 8.6). Its ASCII representation is "00000000".

The answer is that the end of the text in a character array is marked by a special ASCII character, the "NUL" character, which has the ASCII code `00000000`. When we define a character string as in Figure 8.7 we need to be sure to leave room for the longest text it will ever contain *plus one extra element* to hold the trailing NUL character.

Without the NUL character, `printf` would just keep on printing bytes until it happened to find a NUL somewhere in memory or caused the program to crash, since it wouldn't know where the character array ended. In C programs, we represent the NUL character by `\0`.

Each character of a string is stored in memory as ones and zeros, according to the ASCII code. Figure 8.8 shows an example of what you might find in memory if your program contained the statement "`char day[] = "Tuesday";`"

Whenever you use *nano* to create a text file (one of the cpp files you've been writing, for example), the things you type are stored as ASCII-encoded characters in a file on the computer's disk. If you could see the actual bits, and you understood ASCII, you could read the file's contents.

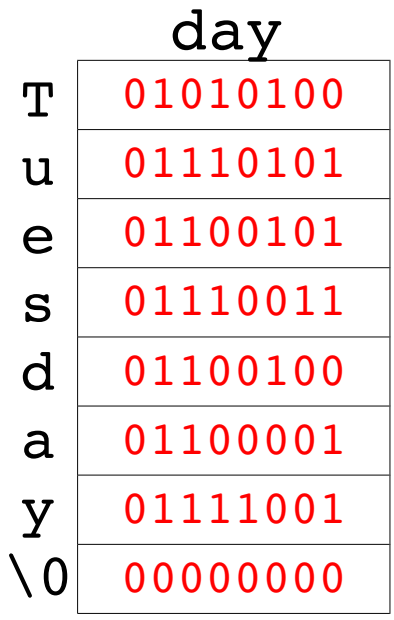


Figure 8.8: This is how the word "Tuesday" would be represented as ones and zeros in memory, stored in an eight-element character array named `day`.

8.6. The `strlen` Function

Can we get our program to tell us the length of a character string? Sure thing! We can use the `strlen` function for this. For example:

```
char name[20] = "Bryan";
int length;
length = strlen(name);
printf ( "This name is %d characters long.\n", length );
```

Some versions of the C compiler might give you an error message if you try to use `strlen` directly as an argument to a function like `printf` or in comparison with an integer in an "`if`" statement. That's because `strlen` doesn't really return an `int` value. Instead of an `int`, `strlen` uses a special data type named `size_t`.

If we tried to write a program containing a statement like this:

```
printf ( "This name is %d characters long.\n", strlen(name) );
```

the C compiler would complain that we've told `printf` to expect an `int` (by using a `%d`), but strlen returns a `size_t`. The complaint would look something like this:

```
program.cpp:6:62: warning: format '%d' expects argument of type 'int',
but argument 2 has type 'size_t {aka long unsigned int}' [-Wformat=]
  printf ( "This name is %d characters long.\n", strlen(name) );
```

The cure for this is to either use a variable like `length`, as we did in the first example above, or to explicitly tell the C compiler to convert `strlen`'s value into an `int`. We could do that like this:

```
printf ( "This name is %d characters long.\n", (int)strlen(name) );
```

We've talked about this kind of re-casting of values in Chapter 2 and Chapter 3.

8.7. Comparing Strings

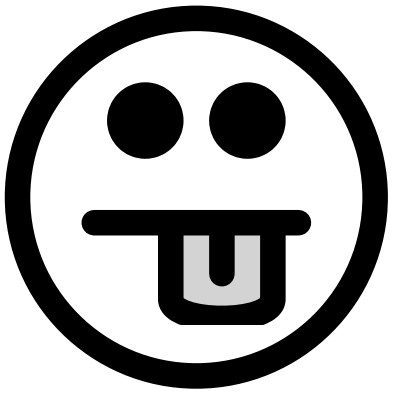
Imagine that we have two character strings, and we want to compare them to see if they're the same. We might try something like the following:

Program 8.2: scomp.cpp (Why doesn't this work?)

```
#include <stdio.h>
int main () {

  char s[] = "junk";
  char t[] = "junk";

  if ( s == t ) {
        printf ("They match.\n");
  } else {
        printf ("They don't match.\n");
  }
}
```



Why doesn't this work? Because "`s`" and "`t`" are arrays. Think about it: if we had two `int` variables, `x` and `y`, we could compare their values with "`if (x==y)`". Similarly, if we had two arrays of `int` elements , `a[10]` and `b[10]`, we could compare two of their elements with "`if (a[1] == b[1])`". But what would we mean if we typed "`if(a==b)`"?

It turns out that, in C, if you type just the name of an array, you get the memory address of the beginning of the array[1]. Since "`s`" and "`t`" in the example above are two different arrays, each of which has its own allocated section of memory, each of them will have a different address. So, "`if(s==t)`" will never be true.

[1] We'll learn more about this later.

If you compile and run Program 8.2 you'll see that it always says "They don't match." This is obviously not the right way to compare two strings.

One way to solve the problem would be to write a "`for`" loop and compare each character in the two strings, one by one. This would be inconvenient though, especially if we had to do it often. Fortunately, C provides us with a function that can compare strings for us. It's called "`strcmp`" (for "string compare").

If we have two character strings, `s` and `t`, and give them to `strcmp` like this:

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

the value of the result will tell us whether the two strings are the same. There are three possiblities:

result = 0	The two strings are identical.
result > 0	`s` is "greater" than `t`
result < 0	`s` is "less" than `t`

In this context "greater than" and "less than" refer to the dictonary order of the two strings. If `s` would come before `t` in a dictionary, `strcmp` says that `s` is less than `t`. According to `strcmp`, "aardvark" is less than "zebra".

Program 8.3 shows the right way to compare two strings.

Program 8.3: scomp.cpp (Doing it the right way.)

```
#include <stdio.h>
#include <string.h>
int main () {
  char s[] = "junk";
  char t[] = "junk";
  if ( strcmp( s, t ) == 0 ) {
        printf ("They match.\n");
  } else {
        printf ("They don't match.\n");
  }
}
```

Notice that we need to add a new `#include` line before we can use the `strcmp` function.

Instead of saying "`strcmp(s,t) == 0`" in our "`if`" statement, we could have saved some typing by saying "`!strcmp(s,t)`". When we say "`if (CONDITION)`", the `CONDITION` is true if it has a non-zero value, and false otherwise. Because `strcmp` returns 0 if the strings are equal, we need to use a ! (read "not") to logically invert this into a true value. You might read such an "`if`" statement as "if `strcmp` *doesn't* return a *non*-zero value...".

Exercise 41: Comparing Strings

Create, compile, and run Program 8.3. Does it do the right thing?

Try changing one of the strings, recompiling, and running again. Does the program properly tell you that the two strings are different now?

8.8. Reading Strings

We've used `scanf` and `fscanf` to read numbers. Now we'd like to use these functions to read text. Can we do it?

There are some complications, and to understand them we'll need to know a little more about how `scanf` and `fscanf` work. Until now, we've taken it on faith that we needed to put an ampersand (`&`) in front of variable names when reading numbers with these functions. The reason that's true is because `scanf` and `fscanf` want the *memory address* of a variable.[2] If I have a variable named `height`, then "`&height`" will be the address of the chunk of memory that the computer has assigned to that variable.

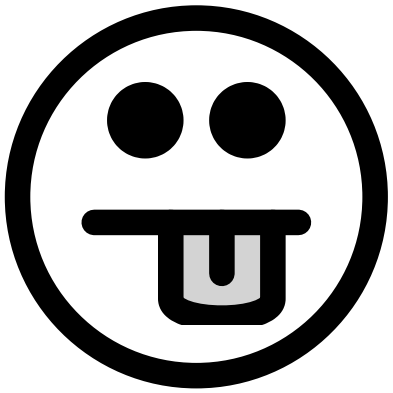
[2] We'll learn why this is so when we study functions in Chapter 9.

As we saw in our bad string comparison example, the name of an array is actually just the memory address of the beginning of the array. This means we can leave off the "`&`" when we read a character array with scanf.

There are still other complications, though, which we can illustrate with Program 8.4. This program asks you to enter some text, and then just tells you what you entered.

Program 8.4: sread.cpp (Not quite getting it right.)

```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  scanf( "%s", string );
  printf( "You said %s\n", string );
}
```



The program defines a character array named `string`, and then uses

`scanf` with the `%s` format specifier to read some text into this array. Notice that the program omits the ampersand we'd use in `scanf` if we were reading a number.

If you try giving this program a word like "Hello", it seems to work fine. In fact, any short, single word will work. But what if we give it something longer, like "`abcdefghijklmnopqrstuvwxyz`"? Then you'll find that the program crashes with a "Segmentation Fault" error. That's because we've tried to go past the end of the `string` character array, which only has room for ten characters. This is the same kind of problem we had with numerical arrays in Chapter 6.

We can fix our program by just adding one letter: change "`%s`" to "`\%9s`" in the `scanf` statement. This tells `scanf` to read *no more than nine characters*. Why nine instead of ten? Because we need to leave room for a NUL character at the end, to mark the end of the string. Now, if we type "`abcdefghijklmnoprstuvwxyz`" the program will print "`abcdefghi`" (just the first nine characters of the text we entered).

Program 8.5: sread.cpp (OK for some things.)

```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  scanf( "%9s", string );
  printf( "You said %s\n", string );
}
```

Note that everything we've said about `scanf` applies to `fscanf` as well.

Exercise 42: Safe String Reading

Create, compile and run Program 8.5. Try giving the program some words without spaces, and then try giving it sentences with spaces in them. Does it behave as expected? What if you type a tab character instead of a space?

There's still one problem left, though. Even the improved version of the program has trouble when we enter text with spaces in it. If we enter "`this is a test`", the program says we typed "`this`".

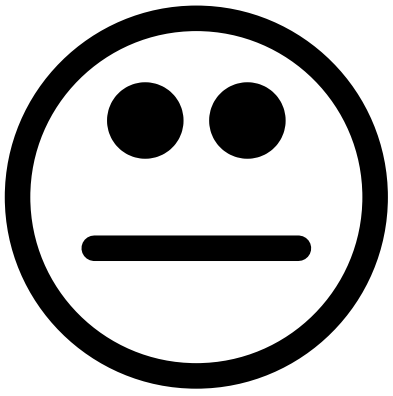
That happens because `scanf` stops reading text (`%s`) when it sees a

"white space" character (a space or a tab). This may not be what you want your program to do. If you need to read strings containing spaces, a better choice is "`fgets`". The `fgets` function reads a specified number of characters from a file. Even though we're reading from the keyboard, not a file, we can still use `fgets`.

Remember that we saw in Chapter 7 that three "files" are automatically opened whenever we run a program: `stdout`, `stderr`, and `stdin`. The first two usually point to your display, and the third (`stdin`) usually points to your keyboard. We can use `fgets` in our program by telling it to read from `stdin`. That's what Program 8.6 does.

Program 8.6: sread.cpp (Better, but see next section...)

```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  fgets( string, 10, stdin );
  printf("You said %s\n",string);
}
```



See next section for caveats.

The `fgets` function takes three arguments: The name of a character string variable in which to store what we read, the size of that character string, and a file handle pointing to an open file to read things from. `fgets` will read, at most, one less than the size of the character string, automatically leaving space for the trailing NUL character.

But what about...?

So why does "`%s`" stop at white spaces? It's so we can do things like this:

```
char name[10];
int year;
printf ( "Enter your last name and birth year: ");
scanf("%9s %d", name, &year);
```

or like this:

```
char firstname[10], lastname[10];
scanf("%9s %9s", firstname, lastname);
```

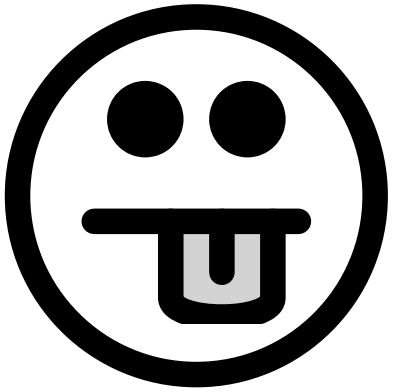
If scanf didn't stop at white spaces, the first example would try to stick things like "Wright 1961" into "name". It would never know

when you were done typing the first word, and had started typing something else.

If you want the things you enter to be broken up into words, `scanf` is a good choice. If you want everything to be put into one variable, `fgets` is the thing to use.

8.9. Line Endings

There's still a potential problem with Program 8.6 though, and it's a subtle one. To illustrate it, let's make a small change to the program and try running it again. The new version is Program 8.7.



Program 8.7: sread.cpp (Watch what happens now...)

```
#include <stdio.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  fgets( string, 10, stdin );
  printf("You said %s. You really did.\n",string);
}
```

If we ran Program 8.7 and entered the text `hello`, we'd see something like the following:

```
Enter some text: hello
You said hello
. You really did.
```

What's going on here? Shouldn't the program have written "You said hello. You really did.", all on one line? The difference is due to the fact that `fgets` interprets the "enter" key as an ASCII "newline" character, and it puts that newline into `string` just like the other characters you typed. In some circumstances that might be OK, but we'll often want to get rid of the extra newline.

Dealing with line endings can be especially tricky if your program reads text from a file. For historical reasons, each of the three most popular operating systems (Windows, OS X, and Linux) uses a different way of indicating the end of a line in an ASCII file. OS X, for example, uses the ASCII "CR" ("Carriage Return") character, which we can write as "`\r`" in C programs. Linux, on the other hand, uses the ASCII "LF"

("Line Feed") character, which we can write as "\n". Windows uses *both*, putting "\r\n" at the end of each line.

To make our programs as portable as possible, it would be nice if they could deal with any of these.

To eliminate such spurious characters we first have to find them. Let's start by looking at a handy C function for finding particular characters in a string. Consider Program 8.8.

Program 8.8: findchar.cpp

```
#include <stdio.h>
#include <string.h>
int main () {
  char welcome[] = "Testing, testing. Are you there?";
  int i;

  i = strcspn( welcome, ".,?" );

  printf("The first punctuation is character number %d\n", i);
}
```

The strcspn function has a name that's hard to remember[3], but what it does is simple. You give strcspn a string and a list of characters you're interested in, then it steps through the string, one character at a time, until it finds an interesting one. When it finds the first interesting character it tells you its location.

[3] It's an abbreviation for "string complementary span", but that's no more memorable.

Program 8.8 defines a character string named welcome. The program uses strcspn to find the location of the first punctuation character in this string. Remember that a character string is just an array of char variables, and that array indices begin with zero. If you start with zero and count characters, you'll find that the "," (the first punctuation mark) is element number 7 of welcome, and that's what Program 8.8 would tell you if you compiled and ran it.

In principle, we could use the strcspn to find \r and \n characters. Once we've found them, we need to know how to get rid of them. That turns out to be easy.

Remember again that a character string is just an array of characters. Once we know which array element holds a letter we want to change, all we need to do is put a different character into that element.

Let's get back to the most recent version of our `sread` program now (Program 8.7). Take a look at Figure 8.9. At the top we see the contents of `string` as Program 8.7 would see it right after the user types "hello" and presses the enter key.

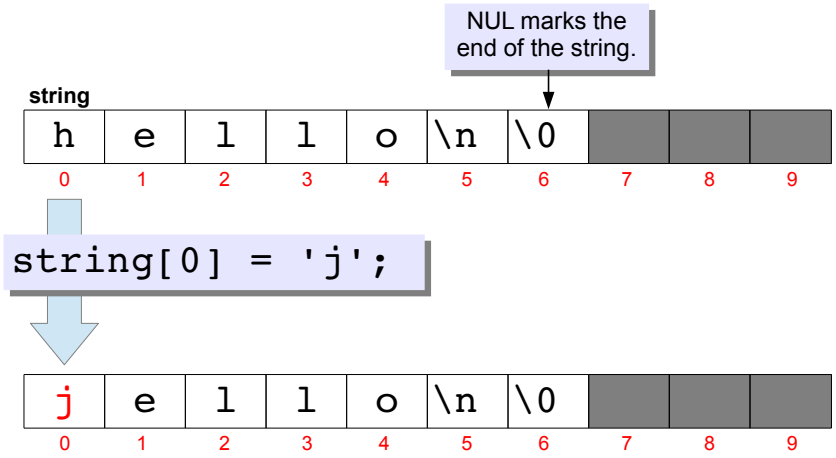


Figure 8.9: Changing one character in a string.

`string` is a 10-element character array. The next-to-last character is "newline", which we represent in C programs as `\n`. Following the newline is an ASCII NUL character, represented by `\0`, which marks the end of the string, as described in Section 8.5 above.

If we wanted to change "hello" into "jello", we could say:

```
string[0] = 'j';
```

making the first letter (element number zero) of `string` a "j" instead of an "h", as shown at the bottom of Figure 8.9.

Now take a look at Figure 8.10. If we wanted to get rid of the newline in `string`, we could replace character number 5. But what should we replace it with? What if we put in another `\0`, as in the bottom of Figure 8.10? Now the newline is gone, and the newly inserted `\0` marks the new end of the string. (The second `\0` is ignored.) We've chopped the troublesome newline off the end of the string!

So, our two-part strategy for removing trailing `\r` and `\n` characters is (1) use `strcspn` to locate them and (2) write an ASCII NUL character

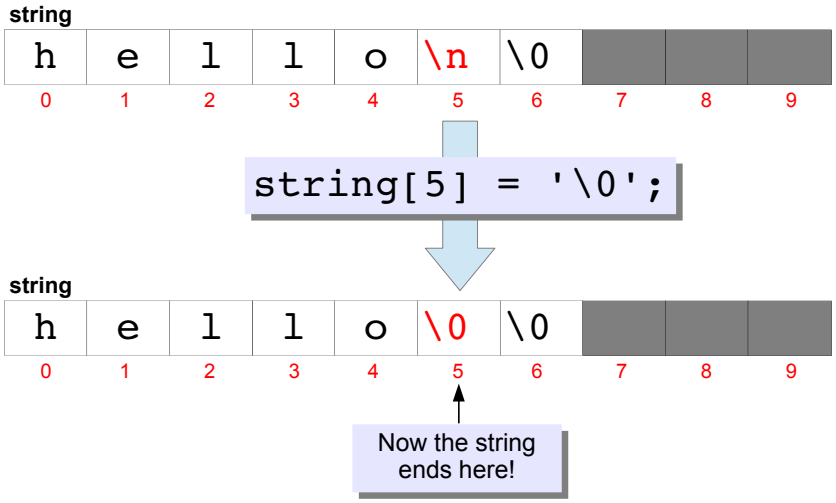


Figure 8.10: Replacing a newline with a NUL.

in their place. Program 8.9 shows a final version of our `sread` program that implements this strategy. As you can see, we only need to add two lines to the program.

Exercise 43: Space, The Final Frontier

Now modify Program 8.5 so that it looks like Program 8.9. Try it again with input that includes spaces or tabs. How does it behave differently?

Program 8.9: sread.cpp (Now deals with spaces and line endings)

```
#include <stdio.h>
#include <string.h>
int main () {
  char string[10];
  printf ("Enter some text: ");
  fgets(string,10,stdin);
  string[ strcspn( string, "\r\n" ) ] = '\0';
  printf("You said %s. You really did.\n",string);
}
```

If we ran Program 8.9 and typed "hello", the result would look like this, as the user would expect:

```
Enter some text: hello
You said hello. You really did.
```

The `strcspn` function gives the location of the first `\r` or `\n`, then the program puts a `\0` at that spot. This would be safe even if the string didn't contain any `\r` or `\n` characters. In that case, `strcspn` returns the location of the `\0` that's already at the end of the string, and the program wouldn't end up changing anything.

It's generally a good idea to use `strcspn` in this way to trim off any extra `\r` or `\n` characters. I recommend you do this whenever you use `fgets`.

Note that in Program 8.9 we could have done things in two explicit steps, by defining an integer variable `i` and saying:

```
i = strcspn( string, "\r\n" );
string[i] = '\0';
```

Either way is fine. Feel free to do it this way if you find it easier to understand.

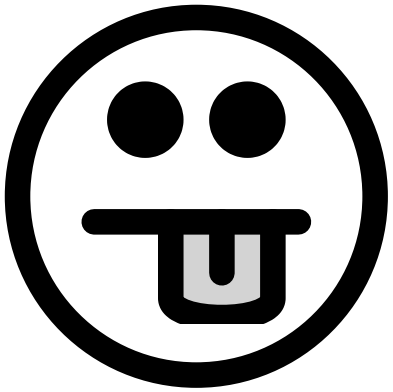
8.10. Assigning Values to Strings

Since strings are arrays, we also need to take care when assigning values to them in our programs. Take a look at Program 8.10 for example. This looks pretty straightforward. We have two character string variables, `s` and `t`, and we want to set `t` equal to `s`, just like we've been doing with numerical variables.

Program 8.10: sassign.cpp (This won't work)

```
#include <stdio.h>
int main () {
  char s[10] = "Testing";
  char t[10];

  t = s;
  printf( "%s\n", t);
}
```



You'll find that *g++* refuses to compile this program though. If you try, you'll probably see an error message like this:

```
sassign.cpp: In function 'int main()':
sassign.cpp:6: error: invalid array assignment
```

Why does this happen? Remember that `t` and `s` are arrays, not single values. The C compiler is telling you that it can't figure out what you want to do here.

What we'd like to do is make each element of the `t` array be the same as the corresponding element of the `s` array. We could write a "`for`" loop to go through all of the array elements and do that, but there's an easier way to do it with character arrays.

We can use the "`spnrintf`" function to "print" the value of one string into another string. This is what we do in Program 8.11.

Program 8.11: sassign.cpp (The right way.)

```
#include <stdio.h>
int main () {
  char s[10] = "Testing";
  char t[10];

  snprintf( t, 10, "%s", s);
  printf( "%s\n", t);
}
```

The `snprintf` function is like `printf`, but it takes two extra arguments: the name of a string, and the number of characters. In Program 8.11 `snprintf` will write a maximum of ten characters into the character string named `t`. It's important that `snprintf` lets us specify the maximum number of characters, so we don't write past the end of `t`.

We could also to things like this:

```
snprintf (t, 10, "Hello world!\n");
```

which would put the text "Hello world!" into `t`.

Internally, `snprintf` just does the same thing as looping through all of the characters in the arrays, one by one, and setting their values.

Exercise 44: For Internal Use Only

Create, compile and run Program 8.11. Try modifying the program by replacing "Testing" with something longer that includes spaces. (You may need to increase the size of the `s` and `t` character arrays.) Recompile the program and make sure it does what you expect.

8.11. Summary of Good String Usage

In the preceding sections we've gone through a bunch of best practices for using character strings. Let's summarize what we've learned:

Comparing Strings

We can't compare strings the same way we compare numbers. If we try to do so, we'll always be misled into thinking that the strings are

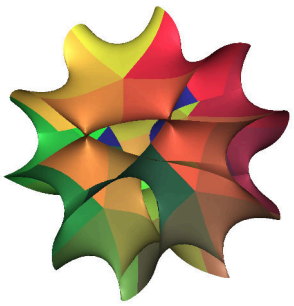


Figure 8.11: Unfortunately, String Theory has nothing to do with character strings, but this Calabi-Yau manifold is too attractive to leave out.

Source: Wikimedia Commons

different, even if they're not. To do it right, use `strcmp` to compare strings. (You'll need to add `#include <string.h>` to use `strcmp`.) Remember that `strcmp` returns zero if the strings are equal. Here's a usage example:

```
if ( strcmp( s, t ) == 0 ) {
    printf( "They're the same!\n" );
}
```

Reading Strings from the User

C provides us with a special format placeholder, `%s`, for reading strings. Since a character string is an array, we need to take care not to go past the end of the array. There are two good ways to read strings: one for when you want each "word" (separated by spaces) to go into its own variable, and another for when you want everything the user types (including spaces) to go into a single variable.

- If you want to split the input wherever there's a space, use `scanf`. Always specify the number of characters by putting a number between `%` and `s`. The number should be one less than the length of the character string array, to leave room for a NUL character at the end. Here's a usage example, suitable for reading text into a 10-character-long string:

  ```
  scanf ( "%9s", string );
  ```

- If you want to put all of the input, spaces and everything, into one variable, use the `fgets` function. Be sure that the size you give it matches the actual size of the character string variable. `fgets` will automatically leave room for the trailing NUL character. Also, use `strcspn` to trim off trailing newlines. Here's a usage example suitable for a 10-character long string:

  ```
  fgets ( string, 10, stdin );
  string[ strcspn( string, "\r\n" ) ] = '\0';
  ```

Assigning Values to Strings

We can't just assign values to character string variables the same way we do with numerical variables. Instead, use snprintf to "print" text into the variable. Here's a usage example that would copy the contents of the variable `s` into the 10-character-long variable `t`:

```
snprintf( t, 10, "%s", s);
```

Writing past the end of a string array is a very common programming bug. It often leads to crashes, and is responsible for many security flaws. Sticking to the methods above will help you avoid these problems in your programs.

8.12. Reading a Gradebook

Let's look at a practical program that uses the string techniques we've been talking about. In this example we'll be reading students' names and grades from a gradebook file and calculating grade averages.

Figure 8.12: Albert Gleizes, *Composition pour Jazz* (1915)
Source: Wikimedia Commons

Take a look a Program 8.12. It reads names and columns of grades from a file like this:

```
Davis          9.2   9.8   9.8   10.   9.2   9.1
Gillespie      8.7   8.7   8.7   8.6   8.9   9.2
Monk           10.   9.0   9.5   9.0   9.1   9.8
Vaughan        9.9   9.9   9.8   8.5   9.0   9.8
Coltrane       9.0   9.1   8.9   9.9   9.7   8.6
Mingus         8.9   9.8   8.6   9.8   9.9   9.8
Parker         9.6   10.   9.1   9.1   9.8   8.9
Holliday       9.2   8.7   10.   8.9   9.8   9.0
Armstrong      8.6   8.6   9.0   9.2   8.6   8.7
Ellington      9.8   9.6   9.6   9.6   10.   10.
Fitzgerald     9.8   9.2   9.9   9.8   8.7   9.6
```

The first column is the student's last name, and the other columns are grades for each of six homework assignments.

Program 8.12 uses `fscanf` to read the students name and store it in the 20-character-long string variable named `lastname`. To make sure it doesn't go past the end of `lastname`, the program tells `fscanf` to use the format "`%19s`", limiting the number of characters to 19 at most, and leaving at least one space to store the terminating NUL character marking the end of the string.

This program uses a technique similar to the one used in our census program in Chapter 7 for reading the multi-column data in the file `grades.dat`. A "`for`" loop reads `ngrades` numbers from each line of the file. Unlike the census program, we don't read the numbers into an array, since this program doesn't care which number was in which column. We only want to add them up, so we can calculate the mean.

The last line of Program 8.12 prints out each student's name and mean grade. Notice that we tell `printf` to print only the first two decimal

Program 8.12: grades.cpp

```
#include <stdio.h>
int main () {
  int ngrades=6;
  char lastname[20];
  double sum, grade;
  int i;
  FILE *gradebook;

  gradebook = fopen("grades.dat","r");
  while ( fscanf( gradebook, "%19s", lastname ) != EOF ) {
                sum = 0.0;
                for ( i=0; i<ngrades; i++ ) {
                 fscanf(gradebook, "%lf", &grade);
                 sum += grade;
                }
                printf ( "%s %.2lf\n", lastname, sum/ngrades );
  }
}
```

Read all lines from file

Get all grades from each line

---

places of the numbers by using "`%.2lf`". As we saw in Chapter 3, a format like "`%`**n**`.`**m**`lf`" means "show m characters with **n** to the right of the decimal place." (We can omit the **n** if we just want to specify the number of decimal places.)

If we ran Program 8.12 we'd see something like this:

```
Davis 9.52
Gillespie 8.80
Monk 9.40
Vaughan 9.48
Coltrane 9.20
Mingus 9.47
Parker 9.42
Holliday 9.27
Armstrong 8.78
Ellington 9.77
Fitzgerald 9.50
```

The program seems to be doing its job, but the output could be more readable. It would be nice if things lined up in straight columns. If we change the last `printf` statement we could make things a little prettier:

```
printf ( "%20s %.2lf\n", lastname, sum/ngrades );
```

We've changed `%s` into `%20s`. If we ran the modified program, the result would look like this:

```
           Davis 9.52
       Gillespie 8.80
            Monk 9.40
          Vaughan 9.48
         Coltrane 9.20
           Mingus 9.47
           Parker 9.42
         Holliday 9.27
        Armstrong 8.78
        Ellington 9.77
      Fitzgerald 9.50
```

What happened? When we say `%20s` we mean "make the output string exactly 20 characters long, padding it on the front with spaces if there's not enough text to fill the full 20 characters."

If we don't like this right-justified style, we can move the text over to the left by changing `%20s` into `%-20s`:

```
Davis                9.52
Gillespie            8.80
Monk                 9.40
Vaughan              9.48
Coltrane             9.20
Mingus               9.47
Parker               9.42
Holliday             9.27
Armstrong            8.78
Ellington            9.77
Fitzgerald           9.50
```

Exercise 45: Reading and Writing Text

Here's a challenge for you. Write a program named `classes.cpp` that asks the user how many classes he or she has on each day of the week. After collecting the data, the program should write the name of each weekday and the number of classes on that day into a data file named `classes.dat`

The program should have a loop that asks the user to enter the name of the day of the week and the number of classes on that day. If the user enters "quit" as the day, the loop should stop.

The program should start out something like this:

```
#include <stdio.h>
#include <string.h>
int main () {
  char day[10];
  int classes;
  FILE *output;
  output = fopen( "classes.dat", "w" );
```

It would be a good idea to use two separate `scanf` statements to read the day name and the number of classes, instead of trying to read both with the same `scanf`. (Can you think of a reason why this is so?)

Here are some hints:

- Remember that you don't need a `&` in front of the variable name when you read a character string with `scanf` (but you do when you read a number).
- You can test to see if a `day` contains the text "quit" like this:

```
if ( strcmp( day, "quit" ) == 0 )
```

- You can write things into a file using `fprintf`, like this:

```
fprintf ( output, "%s %d\n", day, classes );
```

Compile and run your program. The file it creates (`classes.dat`) should look like this:

```
Monday 4
Tuesday 2
Wednesday 3
Thursday 2
Friday 3
```

This is similar to the data files we've graphed with *gnuplot* in the past, except that one of the columns contains text. Start up *gnuplot* and type the following to cause it to use the days of the week as labels on the X axis:

```
set xrange [-1.5:5.5]
set yrange [0:6]
plot "classes.dat" using 2:xticlabels(1) with boxes
```

The first two commands set the range of the X and Y axes so that the data will fit nicely on the graph. The third command tells *gnuplot* to plot the second column of the data, and use the first colum as the labels on the X axis. The result should look something like Figure 8.13.

Sometimes you might want the X axis labels to be vertical. You can do this by giving *gnuplot* the command "`set xtics rotate by 90`", and then typing "`replot`". Give it a try. What happens if you use `-90` instead of `90`?

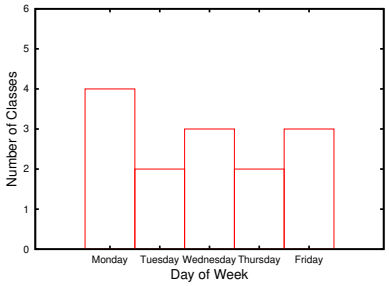


Figure 8.13: Your class schedule might look something like this.

8.13. Reading Column Headers

It would be nice if the columns of our gradebook file had headers, telling the name of each column's assignment. Maybe something like this:

	HW1	HW2	HW3	HW4	HW5	HW6
Davis	9.2	9.8	9.8	10.	9.2	9.1
Gillespie	8.7	8.7	8.7	8.6	8.9	9.2
Monk	10.	9.0	9.5	9.0	9.1	9.8
Vaughan	9.9	9.9	9.8	8.5	9.0	9.8
Coltrane	9.0	9.1	8.9	9.9	9.7	8.6
Mingus	8.9	9.8	8.6	9.8	9.9	9.8
Parker	9.6	10.	9.1	9.1	9.8	8.9
Holliday	9.2	8.7	10.	8.9	9.8	9.0
Armstrong	8.6	8.6	9.0	9.2	8.6	8.7
Ellington	9.8	9.6	9.6	9.6	10.	10.
Fitzgerald	9.8	9.2	9.9	9.8	8.7	9.6

Figure 8.14: Adi Holzer, *Satchmo (Louis Armstrong)* (2002)
Source: Wikimedia Commons

Program 8.13 on page 267 is designed to read this modified data file. In addition to the things our previous program did, this new program also calculates a class average for each assignment. To do this, it needs to sum up the numbers in each column and divide the sum by the

number of students. The sum for each column is stored in an element of the new array named `class_sum`[4]. As the file is read, the students are counted by the variable `nstudents`.

[4] Why do we use `const` when defining `ngrades` here? Look back at page 172 in Chapter 6.

But what about the column headers? Just as we have a class sum for each column, we have a header for each column, and we'd like to save those headers in an array so we can print them out later. But remember that a character string is already an array `char` variables, so we're in need of an *array of arrays*.

That's what the variable named `assignment` is for. It's a six-element array of 10-character strings. Once we've read the column headers into it, we might imagine the array looking like Figure 8.15.

	0	1	2	3	4	5	6	7	8	9
0	H	W	1	\0						
1	H	W	2	\0						
2	H	W	3	\0						
3	H	W	4	\0						
4	H	W	5	\0						
5	H	W	6	\0						
6										

Figure 8.15: An array of character strings holding the column headers from our gradebook file.

Since the column headers are in the first line of the file, they're read first. Program 8.13 uses a "`for`" loop to read the headers into elements of the `assigment` array.

The program then proceeds more or less like Program 8.12, except that the new program also keeps a running sum of each column, in the `class_sum` array, and counts the number of students.

At the end, a new loop goes through all of the assignments, printing out the column header and mean grade for each.

Program 8.13: grades.cpp, Now With Headers!

```
#include <stdio.h>
int main () {
  const int ngrades=6;
  char lastname[20];
  double sum, grade;
  int i;
  double class_sum[ngrades];
  char assignment[ngrades][10];
  int nstudents = 0;
  FILE *gradebook;

  gradebook = fopen("grades-with-headers.dat","r");

  for ( i=0; i<ngrades; i++ ) {
    fscanf( gradebook, "%9s", assignment[i] );
    class_sum[i] = 0.0;
  }

  while ( fscanf( gradebook, "%19s", lastname ) != EOF ) {
    sum = 0.0;
    for ( i=0; i<ngrades; i++ ) {
      fscanf(gradebook, "%lf", &grade);
      sum += grade;
      class_sum[i] += grade;
    }
    printf ( "%-20s %.2lf\n", lastname, sum/ngrades );
    nstudents++;
  }

  printf( "\nClass averages:\n" );
  for ( i=0; i<ngrades; i++ ) {
    printf ( "%10s %.2lf\n", assignment[i], class_sum[i]/nstudents );
  }
}
```

8.14. Handling Errors

Up until now, we've been assuming that the files our programs want to read really exist. But mistakes sometimes happen in the real world. We might accidentally rename or delete a data file, or we might mis-type the file's name when we write it into a program. What happens if a program tries to open a file that doesn't exist? Let's try it and see. Take a look at Program 8.14.

Program 8.14: filecheck.cpp

```
#include <stdio.h>
int main () {
  FILE *input;

  input = fopen( "nosuchfile.dat", "r" );
  // Do some stuff, then close the file...
  fclose ( input );
}
```

If `nosuchfile.dat` doesn't exist, the program will give us an error message saying "`Segmentation fault`"[5]. That's not very helpful, and it might take us a while to figure out that we'd typed the file's name wrong, or put the file in the wrong place.

[5] This error is generated when `fclose` tries to close `input`, which was never really set because the file couldn't be opened.

We can do better. Take a look at Program 8.15. This version of the program checks to see if an error has occurred and prints out a more informative error message. This program does several new things, so let's look at them one by one.

Program 8.15: filecheck.cpp, with error messages

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
int main () {
  FILE *input;

  input = fopen( "nosuchfile.dat", "r" );
  if ( !input )  {
    fprintf ( stderr, "Error opening file: %s\n", strerror(errno) );
    exit(1);
  }

  // Do stuff, then close the file...
  fclose ( input );
}
```

First of all, there are many kinds of errors that a function like `fopen` might encounter. For example:

- Maybe the file you're asking for doesn't exist.
- Maybe you don't have permission to read or create the file.
- If you're trying to create a new file, there might be no more room left on the disk.

In order to tell us what happened, the function identifies each of these conditions with an "error number". Notice that we've added "`errno.h`" to the list of `#include` statements at the top of the program. Among other things, this defines a new variable named `errno` that will always contain a number identifying the most recent error.

Having an error number is a step in the right direction, but words would be even better. That's what the `strerror` function does. It tells us, in plain English, what a particular error number means. `strerror` returns a character string that our program can print out to describe the error. In order to use `strerror` we need to add "`#include <string.h>`".

Finally, we need to have some way to stop the program when we see an error. There's often no point in continuing after something goes wrong, and doing so could even be dangerous. To stop a program immediately, we can use the `exit` function. It takes a single argument (an integer) that's passed along to the operating system to indicate whether the program finished successfully or died because of an error. A value of zero indicates success, and anything else means failure[6]. `exit` requires `stdlib.h`.

[6] We won't make use of these exit values in this book, but they can be handy when writing "scripts" that run programs for you.

If we ran our improved program (with `nosuchfile.dat` still missing), it would say:

```
Error opening file: No such file or directory
```

That's much more informative than "`Segmentation fault`"! When writing programs, think about what might go wrong and try to deal with these situations gracefully.

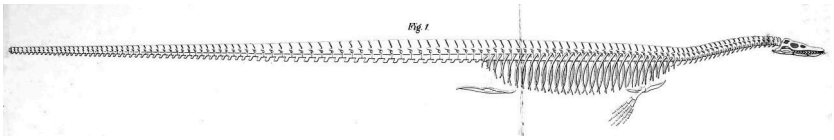


Figure 8.16: We all make mistakes. In 1890, palaeontologist Othniel Marsh humiliated his rival Edward Cope by pointing out that Cope had reconstructed the skeleton of *Elasmosaurus* with the head on the wrong end!

Source: Wikimedia Commons

8.15. Converting Characters to Numbers

As we noted in Section 8.4, the computer stores everything as ones and zeros, and it uses ASCII codes to store characters. For example, the ASCII code for an upper-case `'A'` is `01000001`. If we interpreted this as a binary number, it would be equal to the decimal number 65.

There's an ASCII code for each character on your keyboard, including all the numbers. The ASCII code for the digit `'1'` is `00110001`. Interpreted as a binary number, this would be equivalent to the decimal number 49. Take a look back at Figure 8.6 to see the ASCII codes for some other digits.

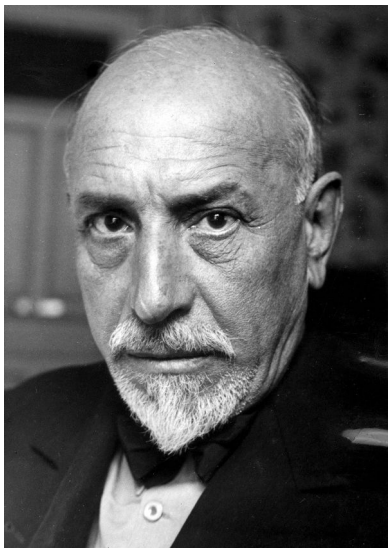


Figure 8.17: Luigi Pirandello was the author of the 1921 play *Six Characters in Search of an Author*. I remember the 1976 PBS production, starring John Houseman and Andy Griffith(!).

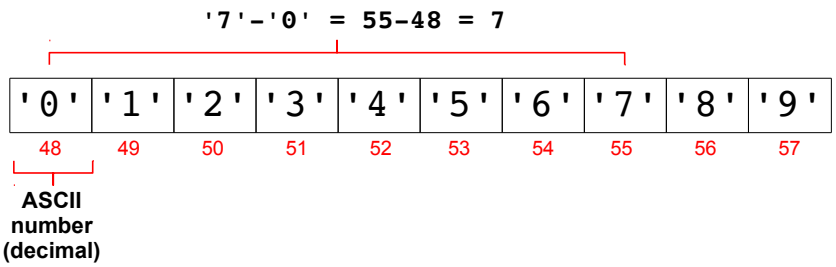
Source: Wikimedia Commons

On the other hand, computers store integer *numbers* as a binary representation of the number. For example, the number `1` would be stored as `00000001`. Maybe you can see how this could create some confusion. As far as the computer is concerned, *character* `'1'` is completely different from the *number* `1`.

Sometimes we'll need to convert a character that represents a digit into an actual number. How can we do that? The first clue is to notice that the ASCII codes for all of the digits in Figure 8.6 are sequential. If we converted these binary numbers into decimal, we'd see that `'0'`, `'1'`, `'2'`, and `'3'` are represented by the numbers 48, 49, 50, and 51.

The second clue is provided by a feature of C that we haven't mentioned before: C is perfectly happy to do math with `char` variables. It just treats the character variable as though it had a value equivalent to the decimal representation of its ASCII code. So, the computer would see `'1'+'2'` as `49+50`, giving a value of `99`.

Using these two clues we can do a little math and determine the numerical value of a character. Take a look at the figure below.



If we want to find the numerical value of the character `'7'` we just need to subtract the character `'0'` from it. In a program, that might look like this:

```
int n;
char c = '7';
n = c - '0';
printf( "The numerical value of %c is %d\n", c, n );
```

But what about...?

What if we have a multi-digit number represented as a string? For example, the string `"186282"`? In principle, we could go through it one digit at a time, converting each character into a number and multiplying it by the appropriate power of ten, then adding up all the results. This would be tedious though, and it seems like something we might need to do pretty often.

Fortunately, as we'll see in Chapter 9, C provides us with two functions that will do the work for us. They're named `atoi` and `atof`. The `atoi` function converts a string of digits into an integer. The `atof` function converts a string that might contain decimal points into a `double`. For example:

```
char ci = "12345";
char cd = "67.890";
int i;
double d;

i = atoi( ci );
d = atof( cd );
```

As we'll see in Chapter 9, these two functions come in very handy in one particular situation: Interpreting command-line arguments.

Let's look at an example that uses this trick.

8.16. Multiplicative Persistence

In Number Theory there's a fun property of numbers called *multiplicative persistence*[7]. Take the number 39, for example. It's represented by the two digits 3 and 9. If we multiply 3×9 we get another number, 27. Multiplying 2×7 gives 14. Multiplying 1×4 gives 4. Now we're down to just one digit after three steps: $39 \rightarrow 27 \rightarrow 14 \rightarrow 4$. We say

[7] See this YouTube video by Matt Parker on the Numberphile channel: https://www.youtube.com/watch?v=Wim9WJeDTHQ

that 39 has a multiplicative persistence of 3, meaning that we can do this procedure of multiplying the digits three times before we get to a single-digit number.

Try this with some other numbers. You'll find that most numbers have only a small persistence. 39 is actually the first one that gets as high as 3. The persistence of 77 is 4. The first number with a persistence of 5 is 679, and you have to go all the way to 6,788 to find a number that has a persistence of 6. Mathematicians think that no base-10 number has a multiplicative persistence greater than 11, but this remains unproven (although it's been checked for numbers up to $10^{20,000}$!).

Figure 8.18: *Still I Persist in Wondering* is the name of an excellent story collection by Edgar Pangborn.
Source: Goodreads

Let's write a program that tests the multiplicative persistence of a given number. Take a look at Program 8.16.

Program 8.16: mpersist.cpp

```
#include <stdio.h>
#include <string.h>
int main () {
  const int maxdigits = 10;
  char number[maxdigits];
  int length;
  int product;
  int i;

  printf ("Please enter a number, up to %d digits long: ", maxdigits-1 );
  fgets ( number, maxdigits, stdin );
  number[ strcspn( number, "\r\n" ) ] = '\0';

  length = strlen( number );
  while ( length > 1 ) {
          product = 1;
          for ( i=0; i<length; i++ ) {
            product *= number[i] - '0';
          }
          snprintf ( number, maxdigits, "%d", product );
          length = strlen( number );
          printf ( "%d %s\n", length, number );
  }
}
```

The program stores a number in a character array. This lets us easily

get each digit of the number, since each digit is one element of the array. The program uses `strlen` to find the string's length. Notice that the "`while`" loop keeps going as long as `length` is greater than one. Each time around the loop, a "`for`" loop goes through all the digits of the number, converting each digit to its numerical equivalent by subtracting `'0'`. The variable named `product` keeps track of the product obtained by multipying the digits together.

If we ran the program, we would see something like this:

```
Please enter a number, up to 9 digits long: 39
2 27
2 14
1 4
```

The first column is the number of digits, and the second column is the current product.

8.17. Pattern Matching

We've seen how `strcmp` lets us compare two strings to see if they're equal, but what if we want to know whether the string fits some fuzzier pattern? For example, we might want to know if the string begins with an upper-case letter, or we might want to check for any of the strings "y", "Y", "yes", or "YES".

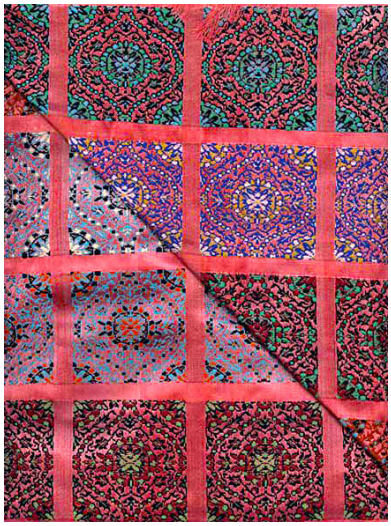


Figure 8.19: *Source: Wikimedia Commons*

The GNU C compiler supports a powerful pattern-matching system called "Regular Expressions". Regular Expressions (sometimes called "regexp" for short) are used in many computer languages. A little knowledge about them will be useful no matter what language you use.

Regular Expressions are a way of specifying a pattern that you want to match. The pattern is written as a group of symbols that can represent particular characters, ranges of characters, or wildcards of various kinds that will match any character. The Regular Expression language is extensive, but here are some commonly-useful symbols and their meanings:

Symbol	**Meaning**
`.`	Match any single character.
`*`	Match zero or more of the preceding item.
`+`	Match one or more of the preceding item.
`?`	Match zero or one of the preceding item.
`{n,m}`	Match at least n, but not more than m, of the preceding item.
`^`	Match the beginning of the line.
`$`	Match the end of the line.
`[abc123]`	Match any of the enclosed list of characters.
`[^abc123]`	Match any character *not* in this list.
`[a-zA-Z0-9]`	Match any of the enclosed ranges of characters.
`this|that`	Match "this" or "that".
`\.`, `\*`, etc.	Match a literal "`.`", "`*`", etc.

Regexp patterns can get confusing very quickly, but here are some simple examples:

`^Y`	Match any string beginning with `Y`.
`^[Bb]ob`	Match any string beginning with `bob` or `Bob`.
`100$`	Match any string ending in `100`.
`^T.*day$`	Match `Tuesday`, `Thursday`, or any other string that begins with a `T` and ends with `day`.
`^data[0-9][0-9]\.dat`	Match `data01.dat`, `data02.dat`, or any other string with `data` followed by two digits and `.dat`.

Program 8.17 uses regular expressions to identify strings that begin with upper-case letters. A string like `Montana` would match, but `montana` wouldn't.

The program uses two functions to accomplish this: `regcomp` and `regexec`. The first function "compiles" a Regular Expression into an internal form that's easier for the computer to use. The second function uses this compiled Regular Expression to test a string. In this case, the Regular Expression we're using is `^[A-Z]`, which matches any string that begins with the upper-case characters `A` through `Z`.

Notice that we need to add `#include <regex.h>` in order to use these functions. `regex.h` also defines a new type of variable, `regex_t`, that's used for storing the compiled version of a Regular Expression.

A complete description of the `regcomp` and `regexec` functions is beyond the scope of this course, but Program 8.17 illustrates their basic usage. In this example, `regcomp` compiles our expression and stores the compiled version in the variable named `reg`. The "`REG_NOSUB | REG_EXTENDED`" argument we give `regcomp` just specifies a couple of options that you'll probably want to use.

The program gives the `regexec` function the compiled Regular Expression (stored in the variable `reg`) and the name of a string variable to test. The other three arguments we give `regexec` aren't really used in this case, but they should usually be set to the values shown here.

Program 8.17: match.cpp

```
#include <regex.h>
#include <stdio.h>
int main()
{
  regex_t reg;
  char string[100];

  printf ("Enter a word: ");
  scanf( "%99s", string );

  regcomp( &reg, "^[A-Z]", REG_NOSUB | REG_EXTENDED );
  if ( regexec( &reg, string, 0, NULL, 0 ) == REG_NOMATCH) {
    printf ("Doesn't match.\n");
  } else {
    printf ("\"%s\" Matches!\n", string);
  }
}
```

8.18. Conclusion

Character strings aren't particularly exciting, but it can be convenient to be able to use them in your programs. When doing so though, be careful not go past the end of your character arrays, and remember that you need to use `strcmp` to compare strings. If you stick to the best practices outlined above, you should be alright.

Figure 8.20: The author's Uncle Buster, playing a stringed instrument. Buster was a character. (Character. String. See what I did there?)

Practice Problems

1. Write a program called `dict.cpp` that asks the user for two words (reading them with `scanf`), and then tells you which word would come first in the dictionary. If the two words are the same, the program should tell you so. Assume the words are less than 100 characters long.

2. Write a program called `hiname.cpp` that asks users to enter their first and last names, on a single line like "Bryan Wright". Use a single `scanf` statement to read the user's names. Make the program then say "Hi", followed by the user's first name, like "Hi Bryan!".

3. Write a program called `getpoem.cpp` that asks users to enter the first line of their favorite poem. Let the line be up to 100 characters long. Make your program open a file named `firstlines.dat` and write the line into the file. Be sure to open the file for "appending", by giving `fopen` an `"a"` as its last argument[8]. Try running the program several times and entering different lines. If you look at `firstlines.dat` with *nano*, you should see all of the lines you've typed in.

4. The following statement will get the current time (measured in seconds since January 1, 1970) and put it into an integer variable named `start`:

   ```
   start = time(NULL);
   ```

   Knowing this, write a program called `type1.cpp` that tests how fast a user can type the phrase "I love programming!". Make sure the program tells the user what to do.**Hints:**

   - Use a character string at least 30 characters long to capture what the user types.
   - Remember to add "`#include <time.h>`" for the `time` function
   - Use `fgets` to read what the user types
   - Check the time before typing and the time after typing, then look at the difference to find out how long it took. Tell the user how many seconds it took him or her to type the phrase.
   - Don't bother to check whether the user typed the right thing. Assume the user is honest.

5. After completing Problem 4 modify the program so that it uses a `for` loop to ask the user to type the phrase three times, then tells

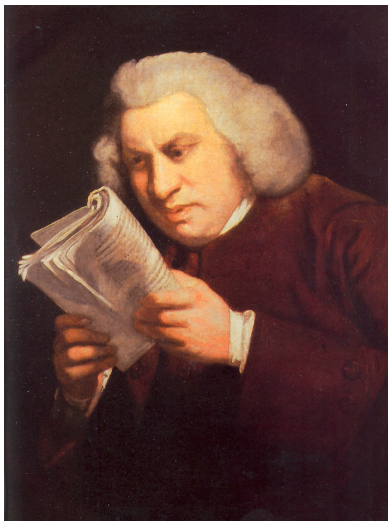


Figure 8.21: In 1755 Samuel Johnson published his *A Dictionary of the English Language*. It remained the most respected English dictionary until more comprehensive dictionaries were published in the 20^{th} Century.

Source: Wikimedia Commons

[8] See Chapter 5.

Figure 8.22: The stylish Olivetti *Valentine* typewriter, designed by Ettore Sottsass to be "sensual and exciting".

Source: Wikimedia Commons

the user his or her average speed. Give the speed in three ways: average number of seconds to type the phrase, number of characters per second, and number of words per minute (where a standard "word" is five characters or spaces). Be sure to allow for non-integer speeds like "1.5 characters per second". Call the new program `typing.cpp`.

Figure 8.23: *Source: Wikimedia Commons*

6. First, fetch a copy of Lewis Carroll's book *Alice in Wonderland* by typing either:

   ```
   wget http://tinyurl.com/y9nrg3xh
   ```

   or

   ```
   curl -L -O http://tinyurl.com/y9nrg3xh
   ```

   After you've downloaded the file, rename it by typing:

   ```
   mv y9nrg3xh alice.txt
   ```

   Then, write a program named `wordlength.cpp` that reads `alice.txt` and reports the average length of the words in the book. The program should define a large (say, 100-character-long) character string, then it should have a loop that repeatedly uses `fscanf` to read words from the file. Use the `strlen` function to find each word's length (see Section 8.6).

   Can you see why we often use 5 characters as the length of a standard "word" when measuring text?

   Hints: To find the average you'll need to first add up the lengths of all the words. I recommend you use a `double` variable to hold this sum. If your program tells you the average word length is exactly an integer, you've done something wrong.

7. After completing Problem 6 create a new version of your program that also makes a histogram that shows the distribution of word lengths. (See examples in Chapter 7.) To do this, you'll need to add an integer array that will keep track of how many words have a given length. Call this array `count` and make it 50 elements long. Each element of the array will contain the number of words that have a length equal to that bin's index. For example, `count[5]` will have the number of 5-letter words. At the end of the program, print out two columns showing the number of letters and how many words had that many letters. Call your new program `wordhist.cpp`.

Notice that, by making our array 50 elements long, we limit ourselves to words with a length between zero and 49 letters. Be sure your program checks the word length to make sure it isn't outside those limits.

If your program also prints out the average word length (as in Problem 6) make sure to put a `#` at the beginning of the line, so *gnuplot* won't be confused by it if you want to plot your results (see Figure 8.24).

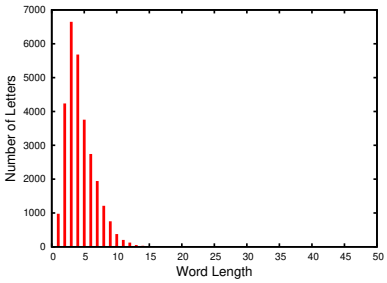


Figure 8.24: If you plot a histogram of the word lengths, you should see something like this.

8. Write a program named `charcount.cpp` that counts how many times each letter of the alphabet appears in a file full of text, treating upper- and lower-case letters as different. Start out by downloading a copy of *Alice in Wonderland* by Lewis Carroll. You can do this with one of the two commands below:

```
wget http://tinyurl.com/y9nrg3xh
```

or

```
curl -L -O http://tinyurl.com/y9nrg3xh
```

After you've downloaded the file, rename it for convenience by typing:

```
mv y9nrg3xh alice.txt
```

Your program should take advantage of the fact that, in C, a character is equivalent to the character's numerical ASCII code. For example, the character "A" is ASCII character number 65. If you have a character variable named `c`, you can get the numerical ASCII code for the character it contains by saying `(int)c` (that is, just "casting" the character as an `int`). These numbers are in the range from zero to 255.

Lewis Carroll, whose real name was Charles Lutwidge Dodgson, was also an accomplished mathematician who made significant contributions to that field.

Source: Wikimedia Commons

At the top of your program, create a 256-element array of integers named `count`, like this:

```
int count[256] = {0};
```

The `= {0}` is a trick we saw earlier in this chapter that sets all of the array elements to zero initially. Your program should contain a "`while`" loop that reads one character at a time from the file `alice.txt`. Each time a character is read, add 1 to the element of `count` that has an index corresponding to that character's ASCII code. You can do that with a line like this:

```
count[ (int)c ]++;
```

After the "`while`" loop is finished, the program should have two "`for`" loops to print its results. The first loop should print counts for character numbers 65 through 90, which corresponds to all the upper-case letters. The second loop should print counts for characters 97 through 122, the lower-case letters. Each line of the output should be printed like this:

```
printf ( "%c %d\n", i, count[i] );
```

where `i` is the character number. Notice that if we print an integer variable using `%c` the program will just print the character corresponding to that number. So, for example, if the file contained the character "A" 807 times, the program would print a line like this for that character:

```
A 807
```

After you've written your program and tested it, try redirecting its output into a file, like this:

```
./charcount > charcount.dat
```

Then you can use *gnuplot* to generate the graph in Figure 8.25. The *gnuplot* command to do this is:

```
plot "charcount.dat" using ($0):2:xtic(1) with impulses
```

There are two bits of magic here: First, `($0)` tells *gnuplot* to use the *line number* as the x value. Second, `xtic(1)` tells *gnuplot* to use the values in column 1 of the data file as the labels on the x axis.

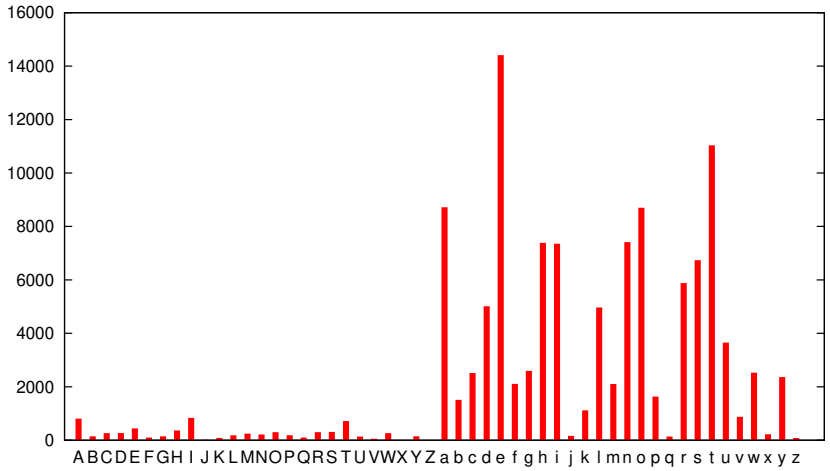


Figure 8.25: Number of each character seen in *Alice in Wonderland*. Notice that "e" is the most common, as is typical in English-language text.

9. First, fetch a copy of the file `unixdict.txt` by typing either:

```
wget http://wiki.puzzlers.org/pub/wordlists/unixdict.txt
```

or

```
curl -L -O http://wiki.puzzlers.org/pub/wordlists/unixdict.txt
```

(whichever works on your computer). This file contains a long list of over 25,000 English words. You can open the file with *nano* to see them.

Figure 8.26: A dictionary from 1st-Century BCE Uruk, in Mesopotamia.
Source: Wikimedia Commons

Write a program named `longestword.cpp` that reads this file and finds the longest word. The program should print the word and its length. Use the `strlen` function to find each word's length (see Section 8.6)

Use the `strerror` function, as described above in Section 8.14, to print an error message if the file `unixdict.txt` can't be found.

Assume that no word is longer than 1,000 characters. Also (of course) assume that all the words have more than zero characters.

Hints: You'll need to define two character strings: one to hold the word you've just read from the file, and another to keep track of the word that has the maximum length so far. Also, you'll find it simpler if you do the following as soon as you read each word:

```
length = strlen(word);
```

then look at the value of `length` when deciding whether this word is longer than the current record-holder. Use an integer variable to keep track of the length of the current record-holder. the top of your program might look something like this:

```
char word[1000];
char maxword[1000];
int length;
int maxlength;
```