Chapter 9: Strings

Notes

- Possible exercise: switch every two characters in a string: “Hello World” = “ehll ooWlrd” or something.
- Possible exercise: write a function to determine if a string contains nothing (is NULL, empty, or blank).
- Escape characters => control characters.

Text is Character Strings

I’ve been preaching numbers and programming for so long you might have forgotten text. Alphabetic characters, punctuation, etc. The truth is that text is represented by *strings* in computer memory *and* logic. A string is a list of values. The term string can be applied to any list of values and a lot more in our real world. If you have beads on a necklace, you may have a string of beads; people standing in line are a string of people, etc. But I’m going to narrow things down a bit. The term string is universally seen as an abbreviation of *character string*, which is a string of characters. From now on when I use the term “string” without context, I am referring to *character* strings.

All the time you’ve been reading written language, you’ve been seeing character strings. Each value in the string is a character; that is each value is not just each letter, but each space, punctuation mark, etc. Typically strings are represented by enclosing them in double quotes:

```
"Hello World"
```

The string above consists of the characters ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘ (space), ‘W’, ‘o’, ‘r’, ‘l’, and ‘d’. The double quotes enclosing the string are not part of the string itself. You’ve probably seen this particular string before, and you’ve been using it and others without realizing, perhaps, that it is a string.

In memory a string is still an array, regardless if it is a string of characters or integers. Each character is actually an index into a character table. This embraces the fact that everything breaks down to a number in computer programming. The character table is also known as a *code page* or *character set*.

Code pages can differ from system to system. However, most systems have code pages that have 256 different indices (0 to 255) and therefore each character in a string will occupy a single byte[1] in memory. You can represent numbers via named values or literals; the same is true with characters. A character literal is just that. However, a

[1] As long as the size of a byte is 8 bits … that too can differ from system to system.

character literal actually represents a numeric index into the current code page: so what you see is *not* necessarily what you get ☺.

Character Literals

To represent a single text character you must surround it in apostrophes, also known as *single quotes*. To use the lower-case character "a"[2] we would write it as such:

```
'a'
```

This literal actually equates to a numerical index in the current code page. The most common code page is either ANSI **<number>** or ASCII. ANSI **<number>** and ASCII are the same for the first 128 indices (0 to 127). A character literal, like 'a' above, can be used any place you would normally use a numeric literal. You could assign it to a variable, use it in a computational expression, or use it with 'cout':

```
int x = 'a';          // assign index value to 'x'
int y = 'a' + 10;     // use in computational expression
cout << 'a' << endl; // output with 'cout'
```

A character literal is interpreted by the compiler as its *numerical index*. Thus it does not actually put 'a' into 'x' above, it puts the index of 'a' from the current character table into 'x'. Unless your computer is in some parallel universe, if you are using a PC then the letter 'a' is actually the number ninety-six (96). Therefore the variable 'x' above will contain ninety-six (96) and 'y' will contain the value one hundred six (106) after the code is executed.

ASCII and Assumptions

I'm going to assume that you are using an ASCII system for any of this to make real sense. That is, a platform (such as operating system and underlying BIOS) and compatible compiler that use the ASCII table for determining which indices correspond to which characters.

I use the term characters because not all of them are letters or numbers. Each individual that you can see in plain text is a single character. This includes spaces, punctuation, and even special characters such as new-lines and tabs. Lower and upper-case letters are even individual characters. A character is just an individual thing, thus a lower-case character is just considered another thing like its conceptually upper-case counterpart. To the computer processor, it sees them as different sets of bits whereas we see them as different forms of the same letter. It is probably for this reason that many languages are case sensitive, because its easier to program it as such and its inevitably faster.

---

[2] I have to assume you know the English alphabet, not only because you have to read my bad English, but there is no identifying *word* for each letter! Aye?

Author's Opinion: Case sensitivity is ridiculous. I think variables should be case sensitive only when it concerns the same variable. I don't think two variables of the same name with different case should be allowed. At the very least compilers should *suggest* possible names that you mis-cased. For example if you write SomeVar instead of someVar, it should say: "Perhaps you meant 'someVar' you knuckle-head!"

My examples will make more sense to you if you are using ASCII because we will have the same results. The letter "A" has an index of 65 in ASCII and if it does on your machine as well then we're working on the same ground. Try this out in a small program:

```
cout << "The index of 'A' is " << (int)'A' << endl;
```

On my system this results in the following output:

```
The index of 'A' is 65
```

If you have the same then joyous day, you're working with an ASCII-compatible system. ANSI <number> is identical to this character set for the first 128 (indices 0 to 127) characters. Luckily for us, that's where all the important ones are. Some systems actually use only 7-bits to store individual characters, or there are formats which expect only 7-bits to be used. This allows for 128 different characters and that's no coincidence.

Char

A character literal has the same sign as the 'char' data type. The name 'char' gives that way almost immediately. Usually character literals are used in conjunction with 'char' variables:

```
char c = 'a';
cout << c << endl;
```

When you output a 'char' variable with 'cout', it will output a character rather than the number stored. That is to say that 'cout' knows that 'char' variables represent indices into the current code page and will output a character from the current one, using the 'char' variable's value as an index.

Don't forget that a 'char' is still an integer-based variable and can store a number. You can use it in computations or pointer arithmetic. An example:

```
char c = 'a';
c++;
cout << c << endl;
```

The above will put the index of the character 'a' into the 'char' variable 'c'. Next 'c' is incremented. Lastly, the character represented by 'c' is printed. The character printed

will not be ‘a’, it will be the character at the next index in the code page. Typically letters are stored sequentially in the code page and the output of this will probably be ‘b’.

Using ‘char’ with ‘cin’, a user can enter an actual character rather than a number. For example:

```
char c;
cin >> c;
```

This would expect the user to input a single *character*, not a number. The ‘cin’ functionality recognizes the ‘char’ data type and therefore expects different input. Many standard functions utilize the ‘char’ data type much differently from the other numeric variables for the sole reason that it typically represents a single text character as well as a number.

If you get nothing else from this chapter, nay this entire book let it be that you understand that *characters are stored as numbers*. Remember I speak in characters not letters. Each numerical digit is actually a character in the code page as well:

```
char c = '0';
```

In the above, the *character* zero is stored into the ‘char’ variable ‘c’. Remember the compiler interprets characters by their index into the code page. The index of the character ‘0’ (zero) in ASCII is 48. Thus, the compiler sees the following which would be processed identically should you write it on an ASCII system:

```
char c = 48;
```

The character “0” (zero) is a representation of the index 48. This is different from the numerical value 0 (zero):

```
char c = 0;
```

That is putting the actual value *zero* (0) into the ‘char’ variable ‘c’. At index zero (0), or the first index, in the ASCII character table we find the NULL character (as explained later). Obviously the NULL character and the character “0” (zero) are not the same. So, please tell me that you can deduce the result of the following condition:

```
if ('0' == 0)
```

If this is boggling your mind, please feel free to take a second, step away from this text, and just ponder it. What we take for granted as text in the computer is merely a list of numbers and below that its only zeroes and ones so be glad I’m dealing with you at this high level! ☺ A computer only understands numbers so things like letters and punctuation must be categorized as characters and each assigned an individual “id” or index. What characters are categorized and at what index is dependent on the character set (code page). In ASCII, there are 256 characters categorized and each given an index of, obviously, zero (0) to two hundred fifty-five (255).

We are not unlike computers. You cannot write something that completely explains a single person, so you use their name. My name is Neil and it is used in writing because there is no letter in any alphabet to sufficiently and completely describe my entire person. Computers are the same way with characters. Just as we cannot use a single letter for each thing in our environment, including ourselves, a computer can only use numbers to describe things. Rather than mentally write in the character itself into its brain, it inscribes its number.

To be done: example program inputting characters.

Character Functions

Although it is nice to all be using the ASCII character set, it is far from being a world-wide custom. In ASCII you know which indices represent which characters simply by looking them up yourself. You could tell if a user entered a numerical digit into a character, by comparing it to one of the ten indices in ASCII that represent digits. Or, you could use the `isdigit()` function which does the same thing, but will work on whatever character set you are compiling for.

This function, and its compatriots, can only be used after including `<ctype.h>` at the top of your program. I'm pretty sure the name of this header file is an abbreviation for "Character Type", alluding to functions dealing with the type of characters.

The `isdigit()` function accepts a single character parameter and returns either a one (1) or a zero (0) depending on if the character passed in is a digit. The syntax of `isdigit()` describes it as accepting an 'int' parameter which sounds unfamiliar where characters are concerned. It is perfectly acceptable, however, to pass in a 'char' variable or character literal:

```
char c = 'a';

if (isdigit('0'))
{
    // this will be executed
}

if (isdigit(c))
{
    // this will not
}
```

All of the character determination functions are like `isdigit()` in that they accept an 'int' parameter pertaining to the character to check and returns an 'int' value of either one (1) (for true) or zero (0) (for false). Since these functions all act like `isdigit()`, I have provided a list of them along with brief descriptions below.

`isalnum`	a letter or a digit
`isalpha`	a letter
`iscntrl`	any control character
`isdigit`	a digit
`isgraph`	any printing character except for the space character
`islower`	a lowercase letter
`isprint`	any printing character
`ispunct`	any punctuation character
`isspace`	a white space character (space, tab, new line, etc.)
`isupper`	an uppercase letter
`isxdigit`	a hexadecimal digit

Because lower and upper-case letters are seen as totally separate characters, there are functions to switch between them. These are `toupper()` and `tolower()` and like the "is" functions above, `<ctype.h>` must be included in order to use them.

Both of these letter conversion functions accept an 'int' parameter corresponding to the character to convert and return an 'int' representing the converted character:

```
char chi = toupper('a');
char clo = tolower(chi);
```

After the above code is executed the 'char' variable 'chi' will contain the character 'A' (or rather the index representing 'A' remember!) and 'clo' will contain the character 'a'.

String Literals and Constants

A string literal is a representation of a constant array of characters. That is, the string can be used in the same places as a value of type 'const char []'. To create a string literal, simply enclose text within double quotation marks, also known as double quotes:

```
"string literal"
```

As mentioned before, everything within the double quotes is part of the string literal, not the quotes themselves. In addition to the characters you've typed, an extra one is automatically added for you: the NULL character. Yes, if you're familiar with pointers then the term should come as no surprise. A NULL character is one with a numeric index of zero. Therefore the literal above contains the characters 's', 't', 'r', 'i', 'n', 'g', ' ' (space), 'l', 'i', 't', 'e', 'r', 'a', 'l', and 0 (null). It also means that the following is equivalent:

```
const char mystr[15] = { 's', 't', 'r', 'i', 'n', 'g', ' ',
                         'l', 'i', 't', 'e', 'r', 'a', 'l', 0 };
```

Notice that the size of the array includes the NULL character. The NULL character is part of the string, it is just unseen. The NULL character is used to know when a string is

ended. When you output a string with 'cout', it loops through all of the subscripts in the string (character array) and outputs each character. It ends the loop when it reaches a NULL character. The NULL character is also known as the *termination character* because it terminates the visible portion of the string.

A character string is a character array and elements may exist in it past the NULL character. Those elements, however, would not be printed when using 'cout' because it stops at the first NULL character. If a NULL character was not use to terminate the string, it would not know when to stop outputting characters and would most likely go past the end of the string into no-strings memory ☺.

Think of a character string as beads on a string. The beads will simply fall off the end of the string unless there is a knot at the end to keep them in. A blind man can determine where the last bead is by the knot that follows it. Likewise, more beads may exist beyond the knot, but where they end cannot be determined unless a knot follows them as well.

Variable Strings

A string literal is a constant array of characters. To create a string that can be changed, you simply create a string variable which is actually just an array of 'char' subscripts. A 'char' array can be used by itself with 'cout' just like a string literal or constant:

```
char mystr[] = { 'H', 'e', 'l', 'l', 'o', ' ',
                 'W', 'o', 'r', 'l', 'd', 0 };
cout << mystr << endl;
```

The output from the above would be:

```
Hello World
```

That's simple enough. As you can see, text is merely several characters that are represented (deep down) by numbers. Luckily, we get to work with character and string literals rather than the numbers themselves.

Once a string is created, it can obviously be modified. Since it is still an array like anything else, you can modify the subscripts individually yourself or call standard helper functions which do common things for you. The latter is much easier, but teaches you less about what is really going on. Thus, helper functions will not be covered until much later in this chapter.

With a variable string the characters that make it up can be changed. For instance, perhaps we want to change the string to "Goodbye World". This sounds like a very simple thing. On paper you would simply erase "Hello" and scribble in "Goodbye". But notice that "Goodbye" occupies more characters than "Hello". Therefore to change "Hello" to "Goodbye" we'll need two (2) extra characters, fourteen (14) in total to store "Goodbye World".

Firstly we'll need to make sure that the string has enough space to store "Goodbye World" which means creating it with fourteen (14) subscripts. This is enough for the thirteen (13) visible characters as well as the terminating NULL character:

```
char mystr[14];
```

We'll start by initializing it to "Hello World" like before:

Now, if the characters of "Goodbye" replace the first characters of 'mystr', the goal has not yet been achieved:

```
mystr[0] = 'G';
mystr[1] = 'o';
mystr[2] = 'o';
mystr[3] = 'd';
mystr[4] = 'b';
mystr[5] = 'y';
mystr[6] = 'e';
```

The string now contains "Goodbyeorld", a far cry from "Goodbye World". Filling in the subscripts does automagically move over the previously existing "World". Unfortunately in this situation your two options both involve setting all the subscripts to store "World". One method would be to simply set *all* the subscripts for the entire "Goodbye World" string. The second would be to *shift* the subscripts storing the " World" characters (including the space) over by two (2):

```
mystr[12] = mystr[10]; // move the 'd' character
mystr[11] = mystr[9]; // move the 'l' character
mystr[10] = mystr[8]; // move the 'r' character
mystr[9] = mystr[7]; // move the 'o' character
mystr[8] = mystr[6]; // move the 'W' character
mystr[7] = mystr[5]; // move the ' ' character
```

Notice anything missing? The NULL character needs to be moved as well, so the following statement would precede the above code:

```
mystr[13] = mystr[11]; // move the NULL character
```

There is a logical reason for performing the shifting *backwards*. That is, I set the subscript values in reverse order from last to first, thirteen (13) to seven (7). If it was done the other way around, the string would not get shifted, it would get garbled. Consider:

```
mystr[7] = mystr[5]; // move the ' ' character
mystr[8] = mystr[6]; // move the 'W' character
mystr[9] = mystr[7]; // move the 'o' character
mystr[10] = mystr[8]; // move the 'r' character
mystr[11] = mystr[9]; // move the 'l' character
mystr[12] = mystr[10]; // move the 'd' character
```

```
mystr[13] = mystr[11]; // move the NULL character
```

The above code looks like it would shift everything to the right properly, doesn't it? It won't though, just look at subscript seven (7). First it is set to the value of subscript five (5) and that completely overwrites whatever previous value was there. Next subscript seven (7) is used to set subscript nine (9). Rather than shifting, both subscript seven (7) and nine (9) have now become the same value: a blank space. In fact, because we're shifting characters by two indices, the resulting string would only have two values (' ' and 'w') repeated over and over, even over the NULL character! This would be a run-time *logic* error rather than a syntax error. The syntax is all correct, but the end result of the logic is not what was intended.

Now, setting subscripts individually is one way of moving the "World" characters over by two, but then that wouldn't have been truly *shifting* the characters. This is a tedious amount of code. Knowing how to use loops effectively with arrays, and therefore strings, lessens the code. The following code would shift the "World" characters as well:

```
int i;
for (i = 13; i > 6; i--)
    mystr[i] = mystr[i - 2];
```

Using loops effectively like this is a bit less apparent and uses an extra variable (i), but is a huge timesaver and the concept can be applied to arbitrary strings. Using loops with pointers to strings is even faster as you'll see later. Again you'll notice that the loop works in reverse from the last character to be set, to the first.

Initializing Strings

Rather than put characters into a 'char' array one at a time, there is an easier method. If you use a constant string (literal or named-value) as an initializer to a character array, the entire contents will be used to initialize the array. All of the following are valid ways of initializing strings using constants or literals:

```
const char hw[] = { 'H', 'e', 'l', 'l', 'o', ' ',
                    'W', 'o', 'r', 'l', 'd', 0 };
char mystr1[12] = "Hello World";
char mystr2[] = "Hello World";
char mystr3[] = hw;
```

These unique assignments cannot happen outside of initialization; or at least will not yield the expected results. It is much like how the initializing of an array or structure cannot be done the same once they are already created.

Pointers to Strings

Pointers can be used to represent strings by pointing to the arrays that hold the characters. The pointer will act identical to the array and also has some additional functionality. Recall that you can use arithmetic on pointers to move through memory quickly and directly. Obviously string pointers can also be dangerous and their purpose is more difficult to detect. A pointer can be NULL as well as point to things other than valid memory or even an array. Consider the following:

```
char mystr[] = "Hello World";
char *p = mystr;
```

The variable 'p' now points to the string "Hello World" which can be accessed by 'mystr'. A character pointer can be used in the same way a character array is used: for input with 'cin', for output 'cout', and for direct modification of each subscript in the array:

```
cout << p << endl;
p[0] = 'h';
cout << p << endl;
```

The above code placed after the earlier sample would yield the output:

```
Hello World
hello World
```

But what if 'p' was actually a pointer to a single 'char' variable rather than an entire string?

```
char c;
p = &c;
```

Using 'p', without dereferencing it, in 'cout' or 'cin' would assume that 'p' points to an array of characters rather than a single character. The danger is very clear, but can easily be avoided. It is also possible to declare a pointer to a literal:

```
char *p = "Hello World";
```

The danger is that the literal should not be modified. It is certainly possible, but the possibility also exists that the data exists in read-only memory and modifying it may cause errors.

Cin and Strings

There are several ways to get user input to strings with 'cin'. The first is the traditional way by using 'cin' with the insertion operator '>>':

```
char str[20];
cin >> str;
```

The immediate issues with this are it only puts the first *word* into the string, and there's no way to limit the number of characters that are inserted. So if you typed in "Hello World", only "Hello" would be stored in 'str'. If you typed "Supercalifragilisticexpialidocious" it would try to put *all* of the characters into 'str'. Obviously something will go wrong because 'str' can only hold twenty (20) total characters (nineteen (19) visible characters and one (1) null character).

There are some functions part of '`cin`' that can be used to get user input to strings. Unfortunately I have to ask you to take a blind leap similar to that of initially breaking into the circle (Chapter 2), because I'm going to introduce some functions that are members of '`cin`' which is an object. I will not cover objects or member functions until later on; please just accept the fact that what I'm about to show you works ☺. The first is '`cin.getline()`' which is a function that accepts a string (through a pointer) followed by an integer[3]:

```
cin.getline(char *dest, int maxchars);
```

The first parameter is the destination for the input (e.g. the character string to put the input into). The second parameter is the maximum number of characters that can be put into the destination string. With the second parameter you can make sure that only the string's memory is touched, nothing past. For example, to input characters into 'str':

```
cin.getline(str, 20);
```

An advantage with this object function ('`getline()`' is a function part of the '`cin`' object) is that you can type in *any* characters in and they will all go into the string. This includes spaces, commas, etc. If you type in "Hello World" then the whole thing will be put into the destination ('str'). If you type in "Supercalifragilisticexpialidocious" then only "Supercalifragilisti" (the first nineteen (19) characters) will be put into 'str'. The remaining characters are simply dropped. Only nineteen (19) characters are copied because the remaining one is reserved for the NULL character (without it the string would be a string of beads with no knot at the end to keep them together ☺).

There is a problem with 'cin.getline' when built on some compilers (***to be done: insert compiler names with this problem***). When you put two calls to '`cin.getline()`' back to back, the second is totally skipped over and a blank string is inserted into the second's destination:

```
char str1[20], str2[20];
cout << "Input first name: ";
cin.getline(str1, 20);
cout << "Input last name: ";
cin.getline(str2, 20);
```

[3] There are more intriguing aspects to this function like a default parameter and overloaded versions; I'm trimming out these things for simplicity's sake.

If you put this into a program, and you're not able to input a last name then your compiler has this problem as well. To fix it you'll need to be the following just after each call to '`cin.getline()`'[4]:

```
while (cin.rdbuf()->in_avail())
    cin.rdbuf()->sbumpc();
```

This hideous code clears out the input buffer.[5] My suggestion to prevent having to write this a lot is to make a '`getline()`' function yourself that automatically clears the buffer after calling '`cin.getline()`':

```
void getline(char *pstr, int max)
{
    cin.getline(pstr, max);
    while (cin.rdbuf()->in_avail())
        cin.rdbuf()->sbumpc();
}
```

Then to input a string, you would simply call:

```
getline(str, 20);
```

Notice the omitting of '`cin.`' because the function called is the one we've written, not the one belonging to '`cin`'. There are other functions for the input of strings, but you will find that '`cin.getline()`' is probably the most useful at this point so I will skip the rest.

The following program makes an example of string input using '`cin.getline()`', clearing the input buffer (to prevent an endless "blank" input), and string output with '`cout`':

```
01  #include <iostream.h>
02
03  void getline(char *, int);
04
05  int main()
06  {
07      char first[32], last[32];
08      cout << "Enter first name: ";
09      getline(first, 32);
10      cout << "Enter last name: ";
11      getline(last, 32);
```

[4] This phenomenon occurs because of the way certain operating systems store new line control data in plain text. Unices and Mac use only a single line-feed (character 10 or 0xA) character, but Windows, DOS, OS/2 and others use a carriage return followed by a line feed (characters 13 and 10 or 0x0D0A). Bad implementations on these systems get data up to the end of the first carriage return *or* line feed rather than a combination of the two. What results is an input buffer that is endlessly blank. The only solution is to explicitly clear up the entire buffer with that ugly code.

[5] The functionality used for this code is straight from the C++ Standard and should work on all modern compilers.

```
12      cout << "You are " << first << " " << last << endl;
13      cout << "(" << last << "," << first << ")" << endl;
14      return 0;
16  }
17
18  void getline(char *pstr, int max)
19  {
20      cin.getline(pstr, max);
21      while (cin.rdbuf()->in_avail())
22          cin.rdbuf()->sbumpc();
23  }
```

Example output of this program is:

```
Enter first name: Neil
Enter last name: Obremski
You are Neil Obremski
(Obremski, Neil)
```

The output obviously depends on what you enter. This program should work fine even on compilers with a bad implementation of '`cin.getline()`' because we clear the input buffer after each call to it (inside the custom function '`getline()`').

String Assignment

Assignment operations do *not* work to copy strings. You can only assign a value to a string when it is initialized. The following will not work:

```
char str[24] = "Hello World";
str = "Goodbye World";
```

And remember the following will not work as you expect:

```
char str[24] = "Hello World";
char *p = str;
p = "Goodbye World";
```

What is 'p' pointing to after the above code is executed? It is pointing to the memory containing the literal string "Goodbye World", it is *not* pointing to 'str' any longer. When you use an assignment operation with a pointer, it changes the pointer value. A pointer's value is a memory address. Thus, when you assign a literal to a pointer, you are simply setting the pointer's value, or memory address, to the address of the literal data. You are not affecting whatever variable the pointer may have been pointing to before. The following causes no problems:

```
char *p = 0;
p = "Hello World";
```

The assignment operation puts a *new* memory address in 'p', it doesn't affect what 'p' *was* pointing to previous to the assignment. The only way to set the value of a string is to set each individual character. There are several ways to do this. You've seen this previously using a loop and by accessing each subscript via an index. Now I'll explain how to do the same thing with a pointer.

Conceptually, character pointer contains the memory address to a single 'char' variable. It can also contain the memory address of the first character in a string of characters. Either way it contains the memory address of a *single character*. If you remember back to pointer arithmetic, you can move the pointer linearly through memory using arithmetic assignment operations:

```
char *p = "Hello World";
p++;
```

After this code is executed, 'p' will be pointing to the 'e' character in the literal string "Hello World". Nothing has happened to the character "Hello World" in memory, the pointer 'p' is simply pointing to the second byte in memory from the start of the "Hello World" memory. This means you can ghost through memory using pointers without affecting it in anyway[6].

Dereferencing a character pointer will yield a single character value:

```
char *p = "Hello World";
p++;
char c = *p;
```

The variable 'c' will contain the character 'e' after the above code is executed. The byte of data at the memory address pointed to by 'p' is copied to the variable 'c'. You can use this set of operations to copy all of the data from one string to another:

```
char str[12];
char *p = "Hello World";
char *wp = str;
while (*pt)
{
    *wp = *pt;
    pt++;
    wp++;
}
*wp = 0;
cout << str << endl;
```

At the end of this code, 'str' will contain the string "Hello World". There are two pointers used: 'wp' and 'pt'. The first, 'wp', is a "write" pointer because values are *assigned* to it when it is dereferenced. The second pointer, 'pt', is a "read" pointer

[6] The arithmetic operations on pointers simply change the numerical address stored in the pointer; it's when you try to dereference the pointer that you try accessing actual memory. Accessing some memory (like address zero) causes an error to happen.

because values are *extracted* from it when it is dereferenced. The loop continues until 'pt' is pointing to the address of a NULL character:

```
while (*pt)
```

After that the character pointed to by 'pt' is copied to the address pointed to by 'wp':

```
*wp = *pt;
```

The pointers are then both incremented to move to the next memory address:

```
pt++;
wp++;
```

If the above lines of code were not present, the pointers would never be pointing to anything except what they were initialized to and the loop would be *infinite* (it could never logically end). These increments cause the characters to be sequentially copied from the literal string into the variable string. The line immediately following the loop caps off the end of the string with a NULL character:

```
*wp = 0;
```

Had this not been there, the string would have no termination character and printing the string will not work as expected (it may even cause a crash). Assignment is copying data from one place to another; thus it is a copy operation. The term copy is used with strings more frequently than the term assignment. It may be the result of the standard string helper functions.

The standard function which does what I just illustrated is called 'strcpy' or "string copy". To use it you must include the file 'string.h' at the top of your program (somewhere around the same block as your '#include <iostream.h>'):

```
#include <string.h>
```

It takes two character point (character string) parameters:

```
strcpy(char *dest, const char *src);
```

The first parameter is a pointer to the destination character string; the destination of the string copy. The second parameter is a pointer to the source character string. Rather than explicitly use character pointers to copy strings, you can use the 'strcpy' function. The previous code (copying "Hello World" into 'str') can be duplicated like so using 'strcpy':

```
char str[12];
strcpy(str, "Hello World");
```

There is no length verification with this function, however. That is, if you declare 'str' to store only twelve (12) characters (eleven (11) visible and one (1) NULL character) and

try to copy in a string longer than that, a problem will occur because memory outside 'str' will be touched. A safer function than 'strcpy' is 'strncpy':

```
strncpy(char *dest, const char *src, const int max);
```

Not only is its first two parameters identical to 'strcpy', but there is an additional parameter specifying the maximum number of characters that should be copied to 'dest'. Following through with previous examples:

```
char str[12];
strncpy(str, "Hello World", 12);
```

This function assures that only a certain number of characters are copied. The maximum number of characters to copy *includes* the NULL character. Thus the call in the code above ensures that only up to eleven (11) visible characters are copied. The NULL character is *always* copied to properly terminate the 'dest' string. Take the following:

```
char str[12];
strncpy(str, "Goodbye World", 12);
```

The string 'str' will contain "Goodbye Wor" (the first eleven (11) visible characters). Note that although not all of the visible characters are copied, the NULL character is still appended to the end; properly terminating the string.

SizeOf String

Getting the size of a character array is not the same as the size of the string. Usually only the characters up to the point of the NULL character are the ones you want to count. The 'sizeof' keyword, on the other hand, represents the size of the entire array or pointer. The size of a string is more commonly referred to as its *length*. The length of a string is the amount of visible characters from the beginning of the string to the NULL character:

```
char str[36] = "Hello World";
int i, length = 0;
for (i = 0; str[i]; i++)
    length++;
cout << "The length of '" << str << "' is " << length << endl;
```

The output of the above will be:

```
The length of 'Hello World' is 12
```

This code tallies the number of visible characters from the first bye of 'str' to its NULL character. The condition used to keep the loop counting is simply 'str[i]'. A NULL character has a numeric value of zero (0). Therefore when the index 'i' represents the NULL character subscript, the condition will be false and the loop will end.

A utility function which does exactly this is ‘strlen’. Its one parameter is a character pointer to a string and its return value is the length of the string passed in:

```
int strlen(const char *str);
```

This function does no checking of the string you pass in. If you pass a NULL string in it will cause a problem.

Author’s Opinion: Unless a function *explicitly* mentions how it handles a NULL parameter, you should not assume that it does.

Concatenation

Appending a string to the end of a string is known as *concatenation*. To do this manually you simply start copying characters to the string starting at the original NULL character. One of the ways to find the NULL character is get a pointer to the string and add the length of the string to it:

```
char str[32] = "Hello World";
char *p = str + strlen(str);
```

After the above code is executed, the pointer ‘p’ will be pointing to the NULL character’s memory address. You can simply start pumping characters in at that point to append to the string:

```
*p = '!';
p++;
*p = '!';
p++;
*p = 0;
cout << str << endl;
```

Don’t forget to end the string with a NULL character. The above will result in the output:

```
Hello World!!
```

The two exclamation points were appended by setting the subscripts at the NULL character and past to the new characters. A new terminating NULL was then set at the point just past the last visible subscript.

The function ‘strcpy’ can be used to append data to a string. This function takes the address of the first byte in a string. To append data to an already-existing string you would simply pass it the address of the NULL character. It would then copy the new string in starting at the NULL character of the original string:

```
char str[32] = "Hello World";
```

```
strcpy(str + strlen(str), "!!");
```

Concatenation of a string can also be done with the more specialized 'strcat()' or *string concatenation* function. This takes two character pointer parameters like 'strcpy', but it automatically finds the end of the first string and attaches the second:

```
void strcat(char *dest, const char *src);
```

Thus, to conclude our example of appending two exclamations to "Hello World":

```
char str[32] = "Hello World";
strcat(str, "!!");
```

Author's Preference: These functions are usually just fine for most circumstances. However, when connecting arbitrary strings I typically use 'strncpy' because I can specify a limit on the number of characters copied. This can prevent the wrong data from being written to.

Comparing Strings

Comparing two strings for equality is not as simple as using the equality operator (==). It can surely be used, but it will not work as expected. When you use the name of a string you are referring to the address of its first character. By using the equality operator you would simply be comparing memory addresses:

```
char str1[]  = "Hello World";
char str2[] = "Hello World";

if (str1 == str2)
{
    // this will never happen because the condition will
    // always be false.
}
```

To effectively compare two strings you must do it at a mind-boggling single character at a time. Because of this, comparing two strings is much more intensive than comparing two simple numbers. With the simple numbers the computer itself can automatically tell you the difference, whereas with strings each character itself is a number and must be tested individually. A standard function for comparing two strings is strcmp() which accepts two strings and returns an 'int':

```
int strcmp(char*, char*);
```

This function will return a negative number if the first string is less than the second, a zero (0) if they are equal, and a positive number if the first string is greater than the second. A string is greater than another if the deciding character (the first different one) between the two has a larger value, and likewise with lesser comparison.

There are some things to note about this. One is that it is case-specific. Two strings will not be the same if their characters are of different casing. Determining equality with this function can be confusing since it returns zero (0), a logical "false", if the strings are equal:

```
char str1[] = "Hello World";
char str2[] = "Hello World";
char str3[] = "Goodbye World";

if (0 == strcmp(str1, str2))
{
    cout << str1 << " and " << str2 << " are equal" << endl;
}
if (0 != strcmp(str1, str3))
{
    cout << str1 << " is not equal with " << str3 << endl;
}
```

The output of the above code would be:

```
Hello World and Hello World are equal
Hello World is not equal with Goodbye World
```

The function strncmp() is identical to strcmp() except that it allows you to specify the maximum number of characters from each string to compare, much like strncpy() is to strcpy().

Another issue with string comparison is deciding whether or not a string contains nothing. There are three ways a string can be in this state. The first is a NULL string or a string whose address is zero. This applies to pointer strings only because a character array will never have a NULL address[7]. The second way is an empty string which is one that begins with a NULL character. Because its first character is NULL, it has a length of zero (0) visible characters. The third and last way is a blank string which is one that contains only white space characters such as spaces and tabs. It is inarguably a string of nothing, but it still has character content.

The first two ways are easy to determine, but the last requires a character by character assessment. Determining a NULL string is as simple as comparing it to zero or NULL itself. Deciding if a string is empty can be done by checking if the first character is NULL or if the string has a length of zero (possibly using `strlen()`). But checking a string for blankness, also a nothing-state, is left up to you as there is no standard function. How would you write a function to determine if a string contained nothing?

Strings and Numbers

[7] Unless it is a *dynamic* array which has not yet been covered.

A popular question is how to convert a string to a number and vice versa. Obviously, writing this yourself would be a gruesome task because you have to compare each character in the string individually, decide which digit and place it represents in the number, and compound it with previous results to eventually get a whole number. I know it takes a bit of work because I wrote some string to number conversion functions when I was first beginning, before I stumbled upon atoi() and eventually strtol().

First, please remind yourself that a string containing a number is completely different from an actual number value. Consider:

```
char str[] = "123";
int x = 123;
```

The numeric value of 'x' above is one hundred and twenty-three (123). It's a single numeric value. The numeric value of 'str' is made up of four values: one for each visible character ('1' == 49, '2' == 50, '3' == 51 in ASCII) and one for the NULL character which has a numeric value of zero (0). This is why conversion is needed. The string must have all of its characters interpreted as part of a whole, single numeric value. You cannot simply assign 'str' to 'x':

```
x = str;
```

Even if you forced that to work through clever casting ('x = (int)str') you would not get the desired results. What would 'x' contain after the assignment operation? It would contain the address of 'str' rather than the number '123'. If you have better luck than a leprechaun the address might actually be the same as that, but instead of relying on chance you can *convert* the string into a number.

Possibly the simplest string to number conversion function is 'atoi()'. To use it and the other functions I will explain here, you must include <stdlib.h>. The name of <stdlib.h> appears to be an abbreviation for "Standard Library". The things provided by this file are very diverse and can have little in common. Anyway, this function atoi() returns an 'int' and accepts a string parameter:

```
int atoi(char *);
```

Therefore to use it you pass it a string and assign the return value to an 'int', or output it, or whatever. Getting back to our previous example:

```
char str[] = "123";
int x = atoi(str);
```

The 'atoi()' function interprets every digit of the string and computes the number it represents. After the above code is executed, 'x' would contain the numeric value one hundred and twenty-three (123). *Voila!* The sibling function 'atol()' works in the same way, but it returns a 'long' value rather than an 'int'; which means it can interpret and return larger numbers on some systems. On most modern systems an 'int' is as big as or larger than a 'long'.

Converting strings to floating point numbers is even more complex than to whole (integer) numbers, but fortunately the helper functions are much the same. The function 'atof()' returns a 'double' which contains the floating point numerical representation of the string passed in. Both this function and its integer mates will return zero if the string cannot be properly interpreted as a number:

```
char str[] = "hello";
int x = atoi(str);
```

The above code will place the value '0' into 'x' because "hello" cannot be converted into an integer. A limitation of 'atoi()' and 'atol()' is that they don't (apparently – not sure what the standard says on this) understand the notations of different numbering systems. Thus if you were to place a hexadecimal number in a string and converted it to a number, you would not get the desired result:

```
char str[] = "0x123";
int num = atoi(str);
```

The above code would yield various results depending on the functions implemented by your C++ software. I would think the most common result would be placing '0' into 'x'; because the parser would see '0' then see 'x' and determine that '0' was the only number in the string. A more advanced parser might ignore the 'x' and you would end up with one hundred and twenty-three (123) in 'num'. But like I said, the results can vary. You should be fairly certain that the string you're converting is actually a number. Unfortunately there are no standard functions to tell if a string can be properly converted to a number, your best bet is just to try.

Advanced functions like 'strtol()' and 'strtod()' exist which allow greater control over the conversion. These, however, contain parameter types that are beyond the pre-requisite of this chapter and I don't want to give you a headache just yet. ☺ Or a second one that is!

Reversing all of these concepts is converting a numeric value into a string of characters. Can you believe that no standard functions exist for this purpose? This is not entirely true, but for the most part there are no functions dedicated to simple conversion from number to string. Note that when you're using "cout" to put data onto the screen, you're converting it to character and string data. The console "screen" is actually a giant grid of characters. Each row is a string and each column is a character. Thus, we can use the same functionality to convert numbers to strings, but instead of printing to the screen we can print to a string.

Fortunately I won't drag your eyeballs through the mud of doing this with 'cout'-style functionality at this point. The simplest way is through the function 'sprintf()' which C programmers will recognize as the work-horse behind 'printf()' (C's version of 'cout'). First off make sure you include <stdio.h> so that you can use it. Now, the syntax of this function is complex so mine below is intentionally simplified:

```
sprintf(char *string, char *format, int number);
```

The ‘format’ parameter is a string that describes how to convert the number into a string. That is, it specifies leading zeroes, trailing zeroes, numbering system, and other fun stuff. The type of the ‘number’ parameter may vary depending on your format string. For now, just stick to ‘int’ variables. For ‘format’ use “%d” when converting ‘int’ variables (or any integer literal). Make sure the string you pass in has enough space to hold the converted number. The most digits a 32-digit integer can use is nine (10) so I usually make sure my string is declared for at least twelve characters (consider a negative sign might be present as well as the NULL character):

```
char str[12];
int x = 123;
sprintf(str, "%d", x);
cout << "str is " << str << endl;
```

The above code converts ‘x’ into a string and copies that into ‘str’. The output would be:

```
str is 123
```

Blarg.

Compound Strings

Because strings are actually sequences of multiple characters, a single string might contain many letters, words, paragraphs, and even pages of text. It is important to keep reminding yourself how string variables work. A string variable represents the address of its first character. You can wield this to your advantage with pointer arithmetic by using *pieces* of strings as if they were whole strings themselves.

How would you generate a string that contained two parts: a string prefix and a numerical suffix; that is, the tail end of the string was added by converting a number to a string. One way would be:

```
int x;
cout << "Enter favorite number: ";
cin >> x;
char str[64] = "User's favorite number is ";
char fav[12];
sprintf(fav, "%d", x);
strcat(str, fav);
```

Or you could have the number converted into a string and copied directly into the whole result using pointer arithmetic and avoid using a separate string altogether:

```
sprintf(str + strlen(str), "%d", x);
```

Can you guess how the above code works? The string 'str' contains a valid string of characters terminated by a NULL character. The variable 'str' itself represents the address of that first character. In the expression we add the result of 'strlen()', which returns the number of visible characters, to 'str' which is the address of its first character. The result of this expression is the address of the NULL character in 'str'; that is, the character just past the last visible character. So, since 'sprintf()' copies its string result, one character at a time, into that address the number is effectively appended to the end.

Now, what if we wanted to divine the number at the end of this string? If we call 'atoi()' with the entire string it will return zero (0) because "User's" is not a number. ☺ Instead we add twenty-six (26), the length of the prefix, to 'str' as we pass it to 'atoi()':

```
x = atoi(str + 26);
```

Thus the function receives the address of the first character of the number converted to a string. It has no way of knowing that the address it receives is not a whole string and no reason to believe otherwise. The function does its job and converts the string it receives (which would only be the number, not the whole string) and returns. As a final touch, let's restore 'str' to its original string value; in a larger program this might be so that a new number can be attached to the end. One way would be to completely re-copy in the original value:

```
strcpy(str, "User's favorite number is ");
```

However, we already know that the last character of this prefix is at index twenty-six (26), so why not just insert a NULL character directly there? This is how that would be done:

```
str[26] = 0;
```

This code would place a numeric zero (0), or NULL character, at the index twenty-six (26). A string is still a character array so this is perfectly valid. Notice how a string takes advantage of both array and pointer syntax. By placing the NULL character here, the remaining characters have been affectively "chopped off". Rather, they still exist but any functions utilizing the string will stop at the terminating NULL character and never look past.

Compound strings would be those that are made of multiple parts and possibly separate pieces of data. These pieces of data can be extracted or modified by manipulating the string appropriately. Strings are sometimes used to contain large blobs of data for this very reason. Rather than deal with a huge structure, you deal with a block of text that must be *parsed*.

String Manipulation

Remember that to modify one "whole" string you must do it one character at a time. A string is a sequence of characters and therefore any modification done to the "string" is actually done to each element in the sequence. Previously I explained how to change the casing of a single character using the '`toupper()`' and '`tolower()`' functions. With this knowledge and the knowledge of strings you should be able to write a similar function that changes the case of the entire string.

String manipulation is a big part of programming because much of how we interact is through language. Computers enjoy their numbers, perhaps a bit too much, but most humans can't communicate properly on a diet of numbers alone. Because of the complexity of utilizing strings of text in computers, there are many standard utilities that come with all modern C++ software to assist you. Some of these you have already seen with the functions I introduced previously.

Other standard helpers to manipulate strings will come along in later in this book. One of the most common factors among high-level languages in the present day is the idea to make strings like other primitive variables. That is, imagine being able to actually *do* the following in C++:

```
string msg = "Hello ";
msg += "World";
cout << msg;
```

The above code lacks the numbers and calculations required of us thus far. In fact, to do the same thing with what I've written it might look like this:

```
char msg[64] = "Hello ";
strcat(msg, "World");
cout << msg;
```

Although strings might seem burdensome in C++, bear with me. I am starting you out at the ground floor and we have not yet reached the escalator. What you have seen is how strings *really* work. But there are many facilities to aid you in string manipulation and many do abstract them into single data types than lists of characters. You may even find that the previous code above is more than just a pipe dream.

Unicode Strings

Understanding different code pages isn't a problem as long as everyone uses ASCII, but everyone doesn't. Languages like Japanese and Russian cannot be utilized with the ASCII character set, they require different (or multiple!) code pages. But again, if you're working on a program only for English (or like languages like Spanish and French which can be displayed using ASCII) this isn't a problem.

Globalization is the process of making one's program usable by a global audience; i.e. those of different languages and customs. There are more things than just language

which make each group of people unique, but that's the only thing I'll cover here. Localization is the process of making one's program usable by a particular audience. If you continue to write programs for Americans (because our English is slightly different from our British brothers) then you are localizing your program to America. Some companies, like Microsoft, build several versions of their programs: a globalized one that is generically sensitive to language and other things and localized ones which are sensitive to customs in specific regions. The localized versions still have the ability to display any other language; they are simply more tailored for a specific region.

Making a program display multiple languages through the normal character systems is fairly complex, because you must switch code pages in order to display different characters for different languages. This switching is beyond the scope of this document, however. I want to discuss the alternative which is Unicode. Unicode is a single gigantic character set that contains every necessary character for every language on the face of the earth … and then some (like smilies and such ☺). Achieving this with normal character literals and variables is quite impossible as they are normally 8 bits on most systems which is only large enough to store one (1) of two hundred fifty-six (256) different values. Thus, Unicode characters are stored in "wide" character variables and literals.

The equivalent of a wide character is a short integer which requires two (2) bytes and has sixty-five thousand five hundred thirty-six (65,536) possible values. So a variable to store these Unicode characters has been around for a long time, but what about literals? At the time of writing this there are still compilers which do not support "wide" character literals. You'll know if yours does if you can precede a literal character with an "L":

```
L'c'
```

This causes the character to be a Unicode character and require two bytes of storage. As luck (and engineers) would have it, the first two hundred fifty-six (256) indices in Unicode are identical to that of ASCII. Thus an ASCII character becomes Unicode simply by putting it into a two byte storage unit rather than a single byte.

Literal wide strings are made in the same way as wide characters: by preceding the string with an "L":

```
L"Hello World"
```

Each character in the string now requires two bytes, so the total amount of storage required for the above string ("Hello World") is twenty-four (24) because of the eleven (11) visible characters and one (1) NULL character. That's right, the NULL character like any other in Unicode requires two (2) bytes.

Although useful for programs destined for multiple cultures, Unicode may not have a place in yours, especially at an early stage in programming. It appears to be a big thing though, so as you move on in your programming career you may see it more and more.