C Strings and Pointers

Motivation

The C++ string class makes it easy to create and manipulate string data, and is a good thing to learn when first starting to program in C++ because it allows you to work with string data without understanding much about why it works or what goes on behind the scenes. You can declare and initialize strings, read data into them, append to them, get their size, and do other kinds of useful things with them. However, it is at least as important to know how to work with another type of string, the C string.

The C string has its detractors, some of whom have well-founded criticism of it. But much of the negative image of the maligned C string comes from its abuse by lazy programmers and "hackers". Because C strings are found in so much legacy code, you cannot call yourself a C++ programmer unless you understand them. Even more important, C++'s input/output library is harder to use when it comes to input validation, whereas the C input/output library, which works entirely with C strings, is easy to use, robust, and powerful.

In addition, the C++ `main()` function has, in addition to the prototype

```
int main()
```

the more important prototype

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

and this latter form is in fact, a prototype whose second argument is an array of C strings. If you ever want to write a program that obtains the command line arguments entered by its user, you need to know how to use C strings.

All of this is why we turn our attention to them here.

Declaring and Initializing C Strings

A C string is an array of characters terminated by a special character called the `NULL` character, also called a `NULL` byte. The `NULL` character is the character whose binary value is 0. One can write the `NULL` character as `'\0'` in a program. Because the `NULL` character takes up one character position in the array, a C string that can store N characters must be declared to have length $N+1$.

Examples

```
char lastname[21];          // can store up to 20 characters, no more
char socsecuritynumber[10]; // can store a 9 digit social security number
char filename[500];         // can store up to 499 characters
```

Unfortunately this little fact seems to be forgotten by many programmers, and when that happens, trouble starts, as you will soon see.

You can initialize a C string in its declaration the same way that you initialize a C++ string, in one of two ways:

```
char lastname[21] = "Ritchie";
char socsecuritynumber[] = "123456789";
```

The first method specifies the length of the array. ***In this case the number of characters in the string literal must be at most one less than this length***. In the second method the compiler determines how large to make the array of characters.

Notice that there is no `NULL` character in either of these initializing declarations. The compiler ensures that the resulting string does have a trailing `NULL` byte.

You will also find declarations of C strings that look like this:

```
char* errmessage = "You did not enter a valid menu choice.";
```

or equivalently

```
char *errmessage = "You did not enter a valid menu choice.";
```

The '`*`' in these declarations means that the variable `errmessage` is a ***pointer***. Later we will talk about pointers. For now, just be aware that both of these declarations declare `errmessage` to be a C string, initialized with the given string, and terminated with a `NULL` byte.

Unlike C++ strings, C strings are of fixed size. They cannot grow automatically by appending more characters to them. They cannot be resized. They are just arrays of characters, nothing more. Later we will see how to change the amount of memory allocated to strings as they need to grow.

Using C Strings

You can treat a C string as an array of characters, using the subscript operator to access each individual character, as in

```
char name[30] = "Thompson";
/* more code here that might change contents of name */
int i = 0;
while ( i < 30 && name[i] != '\0' ) {
    name[i] = toupper( name[i] );
    i++;
}
```

which converts the characters in `name` to uppercase. Notice how the loop entry condition has two conjuncts: the first makes sure that we do not exceed the array bounds and the second looks for the end of the string in case it is shorter than 29 characters (not 30, remember!) In this simple case using a C string is almost the same as using an array. The big difference is that when you have an array, you have to keep track of its length and keep it in a separate variable. With a C string that has been properly initialized, you do not need to know its length because you can search for the `NULL` byte at the end of it.

Searching for the end of a C string is unnecessary because there is a function named `strlen()` in the C `string` library that does this for you. Its prototype is

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

Do not be intimidated by the return type. It is just a `typedef` for `unsigned int`. Sizes are never negative numbers. The argument is a string, expressed not as an array but as a `char*`.

The C string library's header file is `<cstring>` in C++, not `<string>`. The C string library header file is `<string.h>` in C. Therefore, to use the `strlen()` function you need to include `<cstring>`:

```
#include <cstring>
using namespace std;

char name[30] = "Thompson";
/* more code here that might change name */
int i = 0;
while ( i < 30 && i < strlen(name) ) {
    name[i] = toupper( name[i] );
    i++;
}
```

C++ Input and Output

The C++ insertion and extraction operators both work with C strings, when used properly. The following program illustrates:

```
#include <iostream>
#include <cctype>
#include <cstring>
using namespace std;

int main()
{
    char name[10];
    int  i = 0;

    cout << "Enter at most 9 characters: ";
    cin >> name;

    while ( i < 10 && name[i] != '\0' ) {
        name[i] = toupper(name[i]);
        i++;
    }
    cout << name << endl;
    return 0;
}
```

This program will extract the characters entered by the user, until it finds a valid whitespace character or end-of-file character, storing them into the character array name. It will then append a `NULL` character to the end of `name` automatically. The insertion operator `<<` will insert the characters from `name` until it finds a `NULL` character.

There is a great danger with C strings and the extraction operator. It does not check that the input can fit into the string argument. The user can enter more characters than there are room for in the array, and these characters will be stored in the memory positions that logically follow the array of characters, overwriting whatever was there. This is called ***buffer overflow***. Purposefully trying to create buffer overflow is a standard technique used by hackers to try to break into computer systems.

To illustrate, suppose that we enter the string "`abcdefghijkl`" when prompted by the program. It will store these characters into the memory locations starting at `name[0]`, then `name[1]`, `name[2]`, and so on, and continue past the end of the array, storing `k` into the next byte after `name[9]` and `l` into the byte after that. These bytes might actually be part of the storage allocated to the integer `i`, in which case they will overwrite `i`. If the entered string is large enough, the program will crash with a segmentation fault.

To prevent the user from entering more characters than can fit into the array, you can use the I/O manipulator, `setw()` which will limit the number of characters entered to ***one less than its argument***. Remember to `#include <iomanip>` in this case:

```
#include <iostream>
#include <cctype>
#include <cstring>
#include <iomanip>
using namespace std;

int main()
{
    char name[10];
    int  i = 0;

    cout << "Enter at most 9 characters: ";
    cin >> setw(10) >> name;

    while ( i < 10 && i < name[i] != '\0' ) {
        name[i] = toupper(name[i]);
        i++;
    }
    cout << name << endl;

    return 0;
}
```

This program will prevent buffer overflow. If you intend to use the C++ extraction operator as your primary means of reading data into strings of any kind, you should make a habit of using the `setw()` manipulator.

Things You Cannot Do With C Strings (That You Can Do With C++ Strings)

You cannot assign one string to another, as in

```
char name1[10];
char name2[10];
name1 = name2;  // This is an error (invalid array assignment)
```

You cannot assign a string literal to a C string:

```
name = "Harry"; // This is illegal (incompatible types in assignment)
```

You cannot compare two C strings with comparison operators such as <, <=, >, and >= to find which precedes the other lexicographically. For example

```
( name1 < name2 )
```

will simply check whether the starting address of `name1` in memory is smaller than the starting address of `name2` in memory. It does not compare them character by character.

You cannot compare two strings for equality:

```
( name1 == name2 )
```

is true only if they have the same address.

The + operator does not append C strings. It is just an illegal operation.

Pointers, Very Very Briefly

This is a brief introduction to ***pointers***. We need to know a bit about pointers because C strings are closely related to them. Some people tremble at the mention of pointers, as if they are very hard to understand. If you understand reference variables, you can understand pointers.

A pointer variable contains a memory address as its value. Normally, a variable contains a specific value; a pointer variable, in contrast, contains the memory address of another object. We say that a pointer variable, or pointer for short, ***points to*** the object whose address it stores.

Pointers are declared to point to a specific type and can point to objects of that type only. The asterisk `*` is used to declare pointer types.

```
int*    intptr;    // intptr can store the address of an integer variable
double* doubleptr; // doubleptr can store the address of a double variable
char*   charptr;   // charptr can store the address of a character
```

We say that `intptr` is "a pointer to int" and that `doubleptr` is "a pointer to double." Although we could also say that `charptr` is a pointer to char, we say instead that `charptr` is a string. Pointers to `char` are special.

It does not matter where you put the asterisk as long as it is between the type and the variable name– these three are equivalent:

```
int* intptr;
int * intptr;
int *intptr;
```

In general, if `T` is some existing type, `T*` is the type of a variable that can store the address of objects of type `T`, whether it is a simple scalar variable such as `int`, `double`, or `char`, or a more structured variable such as a structure or a class:

```
struct Point { int x; int y; };
Point* Pointptr;
```

The ***address-of*** operator, `&`, can be used to get the memory address of an object. It is the same symbol as the reference operator, unfortunately, and might be a source of confusion. However, if you remember this rule, you will not be confused:

> If the symbol "`&`" appears ***in a declaration***, it is declaring that the variable to its right is a reference variable, whereas if it appears ***in an expression that is not a declaration***, it is the address-of operator and its value is the address of the variable to its right.

```
int x = 972384;
int y = 100;
int &ref = x;  // a declaration, so & is the reference operator
int *p   = &x; // appears on right of =, so it is in an
               // expression and is address-of operator; p stores address of x
p = &y;        // address-of operator; p stores the address of y
int &z = y;    // reference operator; z is an alias for y
```

Dereferencing Pointers

Accessing the object to which a pointer points is called ***dereferencing the pointer***. The "*" operator, when it appears in an expression, dereferences the pointer to its right. Again we have a symbol that appears to have two meanings, and I will shortly give you a way to remain clear about it.

Once a pointer is dereferenced, it can be used as a normal variable. Using the declarations and instructions above,

```
cout << *p;      // p contains address of y, which stores 100,
                 // so 100 is printed. p has been dereferenced.
cin >> *p;       // p is dereferenced before the input operation.
                 // *p is the variable y, so this stores the input into y.
*p = *p + *p;    // just like y = y + y
*p = *p * *p;    // just like y = y * y
```

The dereference operator and the address-of operator have higher precedence than any arithmetic operators, except the increment and decrement operators, `++` and `--`.

If you think about a declaration such as

```
int * p;
```

and treat the `*` as a dereference operator, then it says that `*p` is of type `int`, or in other words, `p` is a variable that, when dereferenced, produces an `int`. If you remember this, then you should not be confused about this operator.

Relationship Between Pointers And Arrays

An array name is a constant pointer; i.e., it is pointer that cannot be assigned a new value. It always contains the address of its first element.

```
int list[5] = {2,4,6,8,10};
cout << *list << endl;       // prints 2 because *list is the contents of list[0]

int * const aPtr = list;     // aPtr has the same address as list
                             // &list[0] is stored in aPtr.
aPtr[2] += 10;               // adds 10 to  list[2]
```

The fact that array names are essentially constant pointers will play an important role in being able to create arrays of arbitrary size while the program is running, rather than by declaring them statically (at compile time).

This is all we need to know for now in order to be able to talk more about C strings and the C string library.

Using The C String Library

The C string library (header file `<cstring>` in C++) contains useful functions for manipulating C strings. On a UNIX system you can read about it by typing `man string.h`. You can also read about it online in many places, such as `http://www.cplusplus.com/reference/clibrary/cstring/`. There are a couple dozen functions defined there, but we will start by looking at the most basic of these. All require the `<cstring>` header file, which is placed into the `std` namespace.

strcmp()

The `strcmp()` function is how you can compare two C strings. Its prototype is

```
int strcmp ( const char * str1, const char * str2 );
```

This compares two C strings, `str1` and `str2` and returns

0	if the two strings are identical, character for character
< 0	if `str1` precedes `str2` lexicographically
> 0	if `str1` follows `str2` lexicographically

"Lexicographically" means that the ascii (or binary) values of the corresponding characters are compared until one is different than the other. If one string is a prefix of the other, it is defined to be smaller.

```
str1      str2      strcmp(str1, str2)     comment
abc       abe       < 0                    (str1 < str2)
Abc       abc       > 0                    (str1 > str2)
abc       abcd      < 0                    (str1 < str2)
```

This little program compares two inputted strings and outputs their relationship:

```
#include <iostream>
#include <iomanip>
#include <cstring>
#include <cctype>
using namespace std;

int main()
{
    char name1[10];
    char name2[10];

    cout << "Enter at most 9 characters: ";
    cin >> setw(10) >> name1;
    cout << "Enter at most 9 characters: ";
    cin >> setw(10) >> name2;

    if ( strcmp(name1, name2) > 0 )
        cout << name1 << " is larger than " << name2 << endl;
    else if ( strcmp(name1, name2) < 0 )
        cout << name1 << " is smaller than " << name2 << endl;
    else
        cout << name1 << " is equal to " << name2 << endl;
    return 0;
}
```

strcat()

The `strcat()` function allows you to append one string to another. Its prototype is:

```
char * strcat ( char * destination, const char * source );
```

This appends a copy of the `source` string to the `destination` string. The terminating `NULL` character in `destination` is overwritten by the first character of `source`, and a `NULL`-character is included at the end of the new string formed by the concatenation of both in `destination`. The `destination` string must be large enough to store the resulting string plus the `NULL` byte, and the two strings are not allowed to overlap. The return value is a pointer to the resulting string.

```
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;

int main()
{
    char name1[100];
    char name2[100];

    cout << "Enter your first name: ";
    cin >> setw(49) >> name1;          // limit to 49 chars
    cout << "Enter your last name: ";
    cin >> setw(49) >> name2;          // limit to 49 chars

    strcat(name1, " ");                // name1 might be 50 bytes now

    cout << "Then you must be " << strcat(name1, name2) << ".\n";
    // name1 might be 50+49+1 = 100 bytes including the NULL byte
    cout << "Nice to meet you, " << name1 << endl;
    return 0;
}
```

Notice that this program uses the result of `strcat()` in two different ways. The first call to it modifies `name1` by appending a space character to it. It ignores the return value. The second call, as the argument to the insertion operator, uses the return value. `name1` is also modified, as the final output statement demonstrates when you run this program.

strcpy()

The `strcpy()` function copies one string into another. Its prototype is

```
char *strcpy(char *destination, const char *source);
```

This copies the source string, including its terminating `NULL` byte, to the destination string. The strings may not overlap, and the destination string must be large enough to receive the copy, including the `NULL` byte. The `strcpy()` function takes the place of the C++ string assignment operator. The following example shows how it can be used.

```
#include <iostream>
#include <iomanip>
#include <cstring>
```

```
using namespace std;


int main()
{
    char lastname[50];
    char firstname[30][10];
    char temp[30];
    int  i = 0, j = 0;

    cout << "Enter your family name: ";
    cin >> setw(50) >> lastname;
    cout << "Enter the names of your children (Ctrl-D to stop): ";
    cin >> setw(30) >> temp;
    while ( !cin.eof() && i < 10 ) {
        strcpy(firstname[i], temp);
        i++;
        cin >> setw(30) >> temp;
    }
    if ( i > 0 ) {
        cout << "Your children are \n";
        while ( j < i )
            cout << firstname[j++] << " " << lastname << endl;
    }
    return 0;
}
```

The `strcpy()` function is used to copy the string stored in `temp` into the array of children's first names. Even though the declaration of `firstname` makes it look like a two dimensional array, and technically it is, it is really an array of C strings, or at least the program treats it as an array of C strings.

Summary

The following table summarizes the differences between the most common string operations using C++ strings and C strings.

C++ string operation	**C string equivalent**
`n = name.size();`	`n = strlen(name);`
`str1 = str2;`	`strcpy(str1, str2);`
`str1 = str1 + str2;`	`strcat(str1, str2);`
`if ( str1 > str2 )`	`if ( strcmp(str1, str2) > 0 )`
`if ( str1 < str2 )`	`if ( strcmp(str1, str2) < 0 )`
`if ( str1 == str2 )`	`if ( strcmp(str1, str2) == 0 )`

Safer Functions

The three functions described above all work in the same way internally. They all need to find the end of the string by looking for the `NULL` byte. This is not a problem when the strings are always `NULL`-terminated, but either for malicious reasons or because of honest mistakes, they can be called with strings that are not `NULL`-terminated, and in this case, many problems can result. Therefore, some time ago, safer versions of these functions were written that take another argument, which is a length argument. They all have the letter 'n' in their names:

```
int   strncmp(const char *str1,  const char *str2,   size_t n);
char *strncat(char *destination, const char *source, size_t n);
char *strncpy(char *destination, const char *source, size_t n);
```

Each differs from its unsafe counterpart in that the third parameter limits the number of characters to be processed. For example, `strncmp()` only compares the first `n` characters of the two strings when given argument `n`, and `strncat()` only appends the first `n` characters of `source` to `destination`, and `strncpy()` only copies the first `n` characters of `source` into `destination`.

More C String Library Functions

strchr()

Sometimes you would like to find a particular character in a C string. The strchr() function lets you do that easily. Its prototype is

```
char *strchr(const char *str, int ch);
```

This returns a pointer to the first occurrence in the string `str` of the character `ch`. Note that it does not return an index, but a pointer! If the character does not occur in the string, it returns a `NULL` pointer. For example,

```
char psswdline[] = "nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin";
char * colonptr = strchr( psswdline, ':' ) ;
```

would set `colonptr` to point to the first colon in the string `psswdline`. One could repeatedly call this function to get all of the positions, but there is a better way to do something like this using `strtok()`.

strspn() and strcspn()

Suppose that you wanted to find the length of the prefix of a string that contains only the characters in a given set of characters. For example, suppose that you wanted to find the longest prefix of a string that contained only the characters `a`, `c`, `g`, and `t`. Of course you could write your own loop, and it would not be very hard (I hope), but there is already a function that does this. Its prototype is

```
size_t strspn(const char *s, const char *accept);
```

`strspn( str, "acgt")` would return the length of the longest prefix of `str` that contained only those characters. If hat length was equal to `strlen(str)`, then that would imply that `str` had no other characters in it.

On the other hand, what if you wanted the length of the longest prefix of `str` that *did not contain* any of the characters in a given set of characters? That is what `strcspn()` is for.

```
size_t strcspn(const char *s, const char *reject);
```

The "c" in the name is for *complement*. Thus, `strcspn(str, ":")` would return the length of the longest prefix of `str` that did not contain a colon. This is something like `strchr()` except that you get a length instead of a pointer to the colon.

strstr()

This function finds substrings of a string. Finding a substring of a string is a harder problem to solve efficiently than the preceding ones. It takes a bit of ingenuity to do it well. For a beginning programmer, this function is handy. If you want to find the first occurrence of the string "`acgaa`" in a string consisting of thousands of occurrences of the characters `a`, `c`, `g`, and `t`, this will do it nicely. Its prototype is

```
char *strstr(const char *haystack, const char *needle);
```

The choice of parameter names is from its man page on my Linux system. `strstr(haystack, needle)` returns a pointer to the first occurrence of the string pointed to by `needle` in the string pointed to by `haystack`. If it has no such substring, `NULL` is returned.

strtok()

The `strtok()` function is different from all of the others in that it can remember the fact that it was called before by your program. The technical way of stating this is that ***it retains state***. It allows you to ***parse*** a string into a sequence of ***tokens***. Parsing a string means decomposing it into its grammatical elements. The individual elements are called tokens in this case. For example, a comma-separated-values file consists of lines each of which is a sequence of tokens separated by commas, as in

```
Harry,Jones,32,photographer,333-33-1234
```

The `strtok()` function will let us break this line into the sequence of tokens

```
Harry
Jones
32
photographer
333-33-1234
```

with very little programming effort. Its prototype is

```
char *strtok(char *str, const char *delim);
```

The first argument is a pointer to the string to be parsed and the second is a pointer to a string consisting of the characters that can act as a delimiter. A delimiter is always a single character. If we want to break up a string on commas, we would use the delimiter string ",". If we want to allow both commas and semicolons, we would use the delimiter string ",;".

`strtok()` remembers its state in the following way. The first time you call it, you give it a pointer to the string you want to parse. Then you repeatedly call it passing a `NULL` first argument instead, and it knows that you still want it to parse the string that you first passed to it. It returns a pointer to a null-terminated string containing the next token on each call until there are no more tokens, when it returns a `NULL` pointer instead. An example will illustrate.

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

int  main(int argc, char *argv[])
{
    char *str, *token, *delim;
    int j = 1;

    if ( argc < 3 ) {
        cerr << "usage: argv[0] string-to-parse  delim-chars\n";
        exit(1);
    }
    str   = strdup(argv[1]);
    delim = strdup(argv[2]);

    token  = strtok(str, delim);
    cout << j << " " << token << endl;

    while ( 1 ) {
        token = strtok(NULL, delim);
        if ( token == NULL )
            break;
        cout << j++ << " " << token << endl;
    }
   return 0;
 }
```

The program expects two command-line arguments: the string to be parsed and a string of delimiters. For example, if we name this program `parseline`, we would use it like this:

```
parseline "Harry,Jones,32,photographer,333-33-1234" ","
```

It would decompose the first string on commas, printing

```
1 Harry
2 Jones
3 32
4 photographer
5 333-33-1234
```

There are several important points to make about `strtok()`.

- It treats adjacent delimiters as a single delimiter. Also, it ignores delimiters at the beginning and end of the string. Thus

  ```
  parseline ,"Harry,Jones,32,,,photographer,,,333-33-1234," ","
  ```

  would produce the same output as the first invocation above. It will never return an empty string. If you need to detect when a token might be empty, you cannot use `strtok()` for this purpose[1].

- It cannot be used on constant strings, meaning strings declared as arrays of chars or with the `const char*` type because it modifies its first argument as it works. Therefore you should copy the string to be parsed into a temporary copy. The `strdup()` function makes a non-constant copy of a string and returns a pointer to it. The program above copies the command-line arguments using `strdup()` and then calls strtok() on the copy.

- The returned token must also be a non-constant string, so it too should be declared as a `char*`.

- If you call `strtok()` with a non-`NULL` first argument, it stops parsing the previous string and starts parsing the new string.

Character Handling: The ctype Library

The C language, and C++ as well, has a rich library of character classification and mapping functions, all of which are declared in the C header `<ctype.h>`, or `<cctype>` in C++. By a mapping function, we mean one that associates one character to another. For example, the function `toupper()` returns the corresponding uppercase letter for a given lower case letter.

All of the functions in the `<ctype>` library take a single character as their argument, but this character is declared as an `int`, not a `char`, and they all return an `int` as well. For example

```
int toupper(int ch);
```

is the prototype for the `toupper()` function. You can pass a character to them, as if they had a `char` parameter, e.g.

```
char ch;
ch = toupper('a'); // store 'A' into ch
```

Although the return type is an `int`, you can still assign the return value to a variable of type `char`. The two useful mapping functions in the library are

[1] You can use `strsep()`, but not all systems have this function.

```
int toupper( int ch);
int tolower(int ch);
```

which return the corresponding character in the opposite case when given a letter, and return the same character as the argument otherwise. Because these functions return an `int`, you have to be a bit careful about how you use them. For example, the instruction

```
cout << toupper('a');
```

prints the code for `'A'` (most likely 65 on your computer) instead of the character `'A'` because it can have an argument of type `int`. If you want to print the `'A'` then either you have to force that return value to be interpreted as a char. You can do this by assigning the return value to a variable of type `char`, as I showed above, or by casting, as in

```
for (char c = 'a'; c <= 'z'; c++ )
    cout << (char) toupper(c);
```

which will print the uppercase letters.

Classification functions take a single character and report on the type of character that it is. There is much overlap in the classifications. The disjoint classes are the lower and upper case letters, digits, space characters, and punctuation. Several functions report on various combinations of those. For example, `isalnum()` reports whether a character is a digit or an alphabetic character. The set of classification functions includes the following:

```
int islower(int ch); // tests whether ch is a lowercase letter
int isupper(int ch); // tests whether ch is an uppercase letter
int isalpha(int ch); // true iff ( islower(ch) || isupper(ch) )
int isdigit(int ch); // tests whether ch is a digit
int isalnum(int ch); // true iff ( isalpha(ch) || isdigit(ch) )
int isblank(int ch); // tests whether ch is a blank (tab or ' ')
int isspace(int ch); // tests for ' ','\t', '\n', '\f', '\r', '\v'
int iscntrl(int ch); // tests for control characters
int isgraph(int ch); // tests whether is printable but not blank
int isprint(int ch); // tests whether ch is printable
int ispunct(int ch); // tests whether ch is punctuation
```

All of these return a non-zero for true and a zero for false. The precise definitions of these functions is different than the comments above; it actually depends upon the ***locale***, which is a collection of environment settings that defines such things as the language, character set, how monetary units are displayed, and so on. The descriptions above are stated for the standard "C" locale, which is what most users have by default.