Chapter

22

C-Style Strings

C++ has two different "string" types; the `string` class from the standard library makes string manipulation easy, but is complex, since it uses dynamic memory. The original "string" type, inherited from the C language, is much simpler.

Though simpler, older C-strings are more **difficult to work with**. Sometimes more efficient, they are also **more error prone**, even **somewhat dangerous**. However, as C++ programmers, you can't ignore them.

Why should you dedicate **any** time to studying C-strings? There are several reasons:

- **Efficiency**. Library `string` objects use dynamic memory and the heap. C-strings are **built into the language**, so you don't need to link library code.
- **Legacy Code**. To **interoperate** with pure C code or older C++ code that predates the `string` type.
- **Library Implementation**. You may want to **implement** your own `string`. Knowing how to manipulate C-strings can greatly simplify this task.
- **Embedded Programming**. Programs written for embedded devices like those in your automobile or toaster, frequently use C-strings.
- **Platform O/S Programming**. For Linux or Windows, you will use C-strings.

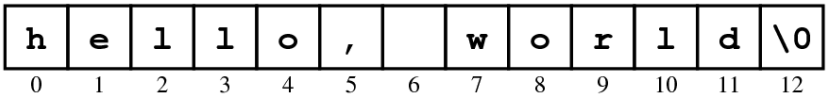
We will encounter many of these cases in the remainder of this course.

C-String Basics

The library `string` type works **as if it were built into the C++ language**. It uses C++ features to allow a `string` to act as a built-in type. C-strings are more primitive:

- C-strings are `char` **arrays** with a **sentinel terminator**, the `NUL` `char` `'\0'`.
- C-strings can be passed to functions without overhead.
- "String literals" automatically include the terminating `NUL`.

The literal `"hello, world"` contains `13` characters—`12` for the meaningful characters plus one extra for the terminating `NUL`. The compiler generates:



C-string functions assume that this **NUL** exists; some insert it for you. Without a **NUL**, functions don't know when the string stops, either returning garbage or crashing. The length of a C-style string is **not stored explicitly**; the **NUL** serves as a **sentinel**.

> *Don't confuse **'0'** with **'\0'**. One is the **ASCII** value **48** and the other **0**.*

How you create a C-string determines **where the characters are stored**.

Character Array C-Strings

To copy characters **into user memory** where they can be modified, write this:

```
const size_t LEN = 1024;    // small strings
char s1[] = "String #1";
char s2[LEN] = "String #2";
```

The C-string **s1** contains **exactly 10** characters; the **9** that appear in "String #1" and the terminating **NUL** character. Space is **allocated on the stack or static storage area**, and the **actual characters are copied** into this "user space". This is shorthand for:

```
 char s1[] = {'S','t','r','i','n','g',' ','#','1','\0'};
```

Because the characters have been copied into memory that you control, you can **change them if you like** using the normal array subscripting operations.

```
s1[0] = 'C';  // OK; all characters are read-write
```

The declaration for **s2** is slightly different. While the effective size of the string is also **9** characters, its allocated size is set by **LEN**, or **1024** in this case. Use **s2** if you want to add information to the end of the string, similar to partially-filled arrays.

Pointer C-Strings

Pointers to **NUL**-terminated character array literals can be used as C-strings:

```
const char *s3 = "String #3";
char *s4 = "String #4";
```

The character array is not copied into your user space. The characters are stored in the **static storage area** when the **program loads**. Attempting to change a character in `s3` is a **compiler error**, because of the `const`.

The declaration for `s4` is obsolete in modern C++, but may be found in older code (and is legal in C). The compiler will probably compile your code (**with warnings**), but **your program will probably crash** when it is run. The portion of the static storage where string literals are stored is **effectively read-only**.

Using C-Strings

C-strings are **not first-class types**: they **do not** work like the built-in types. Look at this example, which tries to **assign**, **compare** and **concatenate** two strings:

```
string str1 = "Hello", str2 = "World";
char cstr1[] = "Hello", cstr2[] = "World";

str1 = "Goodbye";           // assigment OK
cstr1 = "Goodbye";          // ILLEGAL
if (str1 < str2) ...        // comparison OK
if (cstr1 < cstr2) ...      // INCORRECT
str1 += ", ";               // OK
cstr1 += ", ";              // ILLEGAL
```

For the C++ string class, assignment, comparison and concatenation work in the same manner as the built-in types. Use the **assignment operator**, the **relational operators**, and the `+=`. **Not so** for C-strings, where you use functions from the `<cstring>` header.

- `strcpy(dest, src)` is used instead of assignment
- `strcat(dest, src)` is used instead of `+=`
- `strcmp(cstr1, cstr2)` is used instead of the relational operators

In addition, in place of the member function `size()`, you use the `strlen(cstr)` function which returns the number of characters before the `'\0'`.

C-String "Assignment"

Assignment means "copy the thing on the right into the storage on the left". Instead of the assignment operator, used by the built-in types, C-strings use the `strcpy()` function, from the standard library header `<cstring>`, as shown below:

```
const size_t MAX_LEN = 4096;
char dest [MAX_LEN];
// Assume src is a C-style string
strcpy(dest, src);
```

Both `src` and `dest` are C-strings. `strcpy(dest, src)` copies the characters, one by one, from `src` into `dest`, stopping when a `'\0'` is found in `src`. However:

- You don't know if the **actual size** of the C-string `source` is less than `4095` characters (`+1` for the null character). **This code contains a security flaw**.
- You normally won't need anywhere near `4096` characters allocated for `destination`, so **the code is inefficient**.

You must ensure that there is enough space in `dest` to hold a copy of `src`. The icon here does not mean that this code is buggy; instead, it means that the **function itself is intrinsically dangerous**; it's like the symbol found on rat poison.

The function **makes no attempt to check whether the destination has enough room** to hold a copy of the source string. Even if there is not enough memory the function keeps copying, possibly overwriting other data; this called a **buffer overflow**.

The strncpy Function

The (possibly safer, and somewhat controversial[1]) `strncpy()`function copies only **a specified number of characters** from `src` to `dest`.

- Call `strncpy()` with a `dest`, a `src`, and a `count` of characters.
- If the `'\0'` in `src` is found **before** the specified number of characters have been copied, then `strncpy()` will fill the remainder with `'\0'`.
- **You must manually ensure that strncpy appends a terminating null**.

Here is a **semi-safe copy**, given the previous example that avoids overflow (although it doesn't ensure that all of `src` was actually copied; for that you need a loop).

```
dest[MAX_LEN - 1] = '\0';   // pre-terminate
strncpy(dest, src, MAX_LEN - 1);
```

C-String Concatenation

Concatenation is the province of the `strcat()` (unsafe), and the `strncat()` functions, (marginally safer). Here is a (buggy) example:

```
const size_t S_LEN = 10;
char cstr[S_LEN] = "Goodbye";
strcat(cstr, " cruel world!");      // OOPS
cout << strlen(cstr) << " " << cstr << endl;
```

[1] See https://randomascii.wordpress.com/2013/04/03/stop-using-strncpy-already/ for example. Or http://blog.liw.fi/posts/strncpy/ or http://meyering.net/crusade-to-eliminate-strncpy/ or http://udel.edu/~pconrad/UnixAtUD/strcpy.html

When you run, you'll likely see:

```
20 Goodbye cruel world!
```

The C-string **cstr** has room for **9** characters, but you **appear to** have stuffed **21** characters (including the **NUL**), into that smaller space. Not really, of course: **this is a buffer overflow** and the actual **results are undefined**.

The **strncat()** function is marginally safer, if **fairly tricky to use correctly**. If used incorrectly, it overflows just like **strcat()**. Here is the prototype:

```
char * strncat(char *dest, const char *src, size_t count);
```

The tricky part is that **count** is the maximum number of characters to be copied; you must calculate the **correct combined maximum**, before calling the function.

```
const size_t S_LEN = 39;        // max total characters
const cstr[S_LEN + 1] = "This is the intial string";
const char *str2 = "Extra text to add to the string";
strncat(cstr, str2, S_LEN - strlen(cstr));
```

This **isn't efficient** (since you need to count the characters in **cstr** first), but it **does stop copying when the destination string is full**.

***Security Note**: **strncat()** does not check for sufficient space in **dest**; it is therefore a potential cause of buffer overruns. Keep in mind that count limits the number of characters appended; it is not a limit on the size of **dest**.*

Comparing C-Strings

Do not use the relational operators (**<**, **==**, etc.) to compare C-strings. Instead, use the library function **strcmp()**, which compares **s1** and **s2** **lexicographically** and returns an integer indicating their relationship:

- **Zero** if the two strings are equal.
- **Negative** if the first string lexicographically precedes the second string. (Lexicographically simply means "in dictionary order").
- **Positive** if the first string lexicographically follows the second string.

Here's a quick example. The C-strings **s1** and **s2** are initialized somewhere else. Since we don't need to modify either argument, we can use "pointer-style" C-strings.

```
const char *s1 = ..., *s2 = ...;

int result = strcmp(s1, s2);

if (result == 0) ...            // equal
else if (result < 0) ...        // s1 < s2
else ...                        // s1 > s2
```

To use **strcmp()** correctly:

- Call the function and save the **int** it returns.
- Use the returned value with a relational operator.
- **Don't** treat the return value from **strcmp()** as a Boolean expression.
- Don't repeatedly call **strcmp()** on the same strings (inefficient).

Writing C-String Functions

Let's look at several implementations of the standard library functions beginning with **strlen()**. To find the length, you count characters until you reach the **'\0'**. Here is an implementation that uses array notation.

```
size_t strlen(const char str[])
{
    size_t len = 0;
    while (str[len] != '\0') len++;
    return len;
}
```

Note that the return type **must be** ***size_t*** *(not* ***int****), because we can't have a negative length on a string. The array must be* ***const****, otherwise it* **would be illegal** *to call the function using a C-string literal.*

Pointer Difference Version

It is more efficient to advance the pointer until it reaches the end and then to use pointer subtraction (or pointer difference) to determine the number of characters:

```
size_t strlen(const char *str)
{
    const char *cp = str;
    while (*cp != '\0') cp++;
    return cp - str;
}
```

Implementing `strcpy()`

The **`strcpy`** function is often **even more cryptic** than **`strlen`**.

```
char * strcpy(char *dest, const char *src)
{
    char *result = dest;
    while (*dest++ = *src++);
    return result;
}
```

This **very, very common idiom** has so many potential pitfalls, that it is likely that your IDE will mark it with a warning. Although technically not incorrect, it is intrinsically dangerous code, since a small mistake can break the loop entirely.

- The **body of the `while` loop is empty**; all of the work occurs in the extremely streamlined test expression: **`*dest++ = *src++`**
- This expression is **not a comparison**, but an **embedded assignment**. If you accidently use a comparison, the loop will not work.
- This copies the character addressed by **`src`** into the address indicated by **`dest`**, incrementing each pointer. **If you use prefix increment instead of postfix, this does not work.**
- The result is zero—and therefore **`false`**—only when the code copies the **`NUL`** character at the end of the string.

Implementing `strcmp()`

Like **`strcpy()`**, most implementations of **`strcmp()`** are cryptic. Here's GNU C:

```
int strcmp(const char *s1, const char *s2)
{
    const unsigned char *a1, *a2;
    for (a1 = reinterpret_cast<const unsigned char *>(s1),
         a2 = reinterpret_cast<const unsigned char *>(s2);
         *a1 == *a2; a1++, a2++)
        if (*a1 == '\0') return 0;
    return *a1 - *a2;
}
```

This version returns the difference between the first two mismatched characters. `a1` and `a2` are pointers to `unsigned char`, so the characters can be interpreted as raw values between `0-255`, specified by employing a `reinterpret_cast`.

```
int strcmp(const char *s1, const char *s2)
{
    for ( ; *s1 == *s2; s1++, s2++)
        if (*s1 = '\0') return 0;
    return ((*(unsigned char *)s1
            < *(unsigned char *)s2) ? -1 : +1);
}
```

Here is an alternate (Apple/Next/PPC) version of the same function, which returns `0`, `+1` and `-1` instead of the difference. This function, written in 1992, uses old-style C casts to handle the signed/unsigned problem, similar to those found in Java.

Writing Your Own Functions

To write your own C-String functions you can use **either** array notation or pointer notation, whichever you find more convenient; **neither** is more efficient than the other. The things you need to remember are:

- Find the `NUL` character in the string. All C-String functions rely on this.
- Preserve the `NUL` character in the string. It is up to you to make sure that any destination strings are correctly terminated.

To make this more concrete, let's look at a couple of examples.

Finding the First Occurrence of a Character

Here's the algorithm you would use:

```
Loop through a string until the NUL character
   If current character is the target
      Return its index
Return the error code
```

Assume that we use -1 for the error code. An array-notation implementation of the function could look like this:

```
int find(const char a[], char target)
{
    for (int i = 0; a[i] != '\0'; ++i)
        if (a[i] == target) return i;
    return -1;
}
```

A (more cryptic) pointer-notation implementation might look like this:

```
int find(const char* s, char target)
{
    auto *p = s;
    while (*p && *p != target) p++;
    if (*p) return p - s;
    return -1;
}
```

The **temporary pointer** `p` is moved through the C-string `s`. The expression `*p` is `false` when the `NUL` is encountered. Since the loop **must end** when you encounter the `NUL`, or, when you find the target, you know that the loop **terminates in every case**.

After the loop is over, there are **two** possibilities. If `p` is pointing at **any** character, it **must** be the `target` character. That means you can use **pointer difference** to return the index. Otherwise, `p` **must** be pointing at the `NUL` character and you can return -1.

Finding the Last Occurrence of a Character

You might think that the easiest thing would be to **start at the back of the string** and then loop towards the front. That's what you'd do with a C++ `string`. However, with C-strings, you can't find the length **without first looking at every character**, so looping backwards is actually more inefficient than simply going forward.

Here's an efficient **array-notation** implementation of the function:

```
int find(const char a[], char target)
{
    int result = -1;
    for (int i = 0; a[i] != '\0'; ++i)
        if (a[i] == target)
            result = i;
    return result;
}
```

Finding One String Inside Another

This is more easily done **using several pointers**. Here's the algorithm:

```
String search, string target
Pointer p set to search
While *p
   Pointer p1<-p
   Pointer p2<-target
   While *p1 && *p2 && *p1 == *p2
      p1++, p2++
   If *p2 == '\0' return p - search
   p++
Return the error code
```

Go ahead and implement this yourself with this CodeCheck problem.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **`make submit`**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.