17.10 — C-style strings

In lesson 17.7 -- Introduction to C-style arrays, we introduced C-style arrays, which allow us to define a sequential collection of elements:

```
int testScore[30] {}; // an array of 30 ints, indices 0 through 29
```

In lesson 5.2 -- Literals, we defined a string as a collection of sequential characters (such as "Hello, world!"), and introduced C-style string literals. We also noted that the C-style string literal "Hello, world!" has type `const char[14]` (13 explicit characters plus 1 hidden null-terminator character).

If you hadn't connected the dots before, it should be obvious now that C-style strings are just C-style arrays whose element type is `char` or `const char`!

Although C-style string literals are fine to use in our code, C-style string objects have fallen out of favor in modern C++ because they are hard to use and dangerous (with `std::string` and `std::string_view` being the modern replacements). Regardless, you may still run across uses of C-style string objects in older code, and we would be remiss not to cover them at all.

Therefore, in this lesson, we'll take a look at the most important points regarding C-style string objects in modern C++.

Defining C-style strings

To define a C-style string variable, simply declare a C-style array variable of `char` (or `const char` / `constexpr char`):

```
char str1[8]{};                    // an array of 8 char, indices 0 through 7

const char str2[]{ "string" };     // an array of 7 char, indices 0 through 6
constexpr char str3[] { "hello" }; // an array of 6 const char, indices 0 through 5
```

Remember that we need an extra character for the implicit null terminator.

When defining C-style strings with an initializer, we highly recommend omitting the array length and letting the compiler calculate the length. That way if the initializer changes in the future, you won't have to remember to update the length, and there is no risk in forgetting to include an extra element to hold the null terminator.

C-style strings will decay

In lesson 17.8 -- C-style array decay, we discussed how C-style arrays will decay into a pointer in most circumstances. Because C-style strings are C-style arrays, they will decay -- C-style string literals decay into a `const char*`, and C-style string arrays decay into either a `const char*` or `char*` depending on whether the array is const. And when a C-style string decays into a pointer, the length of the string (encoded in the type information) is lost.

This loss of length information is the reason C-style strings have a null-terminator. The length of the string can be (inefficiently) regenerated by counting the number of elements between the start of the string and the null terminator. Alternatively, the string can be traversed by iterating from the start until we hit the null terminator.

Outputting a C-style string

When outputting a C-style string, `std::cout` outputs characters until it encounters the null terminator. This null terminator marks the end of the string, so that decayed strings (which have lost their length information) can still be printed.

```
#include <iostream>

void print(char ptr[])
{
    std::cout << ptr << '\n'; // output string
}

int main()
{
    char str[]{ "string" };
    std::cout << str << '\n'; // outputs string

    print(str);

    return 0;
}
```

If you try to print a string that does not have a null terminator (e.g. because the null-terminator was overwritten somehow), the result will be undefined behavior. The most likely outcome in this case will be that all the characters in the string are printed, and then it will just keep printing everything in adjacent memory slots (interpreted as a character) until it happens to hit a byte of memory containing a 0 (which will be interpreted as a null terminator)!

Inputting C-style strings

Consider the case where we are asking the user to roll a die as many times as they wish and enter the numbers rolled without spaces (e.g. `524412616`). How many characters will the user enter? We have no idea.

Because C-style strings are fixed-size arrays, the solution is to declare an array larger than we are ever likely to need:

```
#include <iostream>

int main()
{
    char rolls[255] {}; // declare array large enough to hold 254 characters + null
terminator
    std::cout << "Enter your rolls: ";
    std::cin >> rolls;
    std::cout << "You entered: " << rolls << '\n';

    return 0;
}
```

Prior to C++20, `std::cin >> rolls` would extract as many characters as possible to `rolls` (stopping at the first non-leading whitespace). Nothing is stopping the user from entering more than 254 characters (either unintentionally, or maliciously). And if that happens, the user's input will overflow the `rolls` array and undefined behavior will result.

Key insight

Array overflow or **buffer overflow** is a computer security issue that occurs when more data is copied into storage than the storage can hold. In such cases, the memory just beyond the storage will be overwritten, leading to undefined behavior. Malicious actors can potentially exploit such flaws to overwrite the contents of memory, hoping to change the program's behavior in some advantageous way.

In C++20, `operator>>` was changed so that it only works for inputting non-decayed C-style strings. This allows `operator>>` to extract only as many characters as the C-style string's length will allow, preventing overflow. But this also means you can no longer use `operator>>` to input to decayed C-style strings.

The recommended way of reading C-style strings using `std::cin` is as follows:

```
#include <iostream>
#include <iterator> // for std::size

int main()
{
    char rolls[255] {}; // declare array large enough to hold 254 characters + null
terminator
    std::cout << "Enter your rolls: ";
    std::cin.getline(rolls, std::size(rolls));
    std::cout << "You entered: " << rolls << '\n';

    return 0;
}
```

This call to `cin.getline()` will read up to 254 characters (including whitespace) into `rolls`. Any excess characters will be discarded. Because `getline()` takes a length, we can provide the maximum number of characters to accept. With a non-decayed array, this is easy -- we can use `std::size()` to get the array length. With a decayed array, we have to determine the length in some other way. And if we provide the wrong length, our program may malfunction or have security issues.

In modern C++, when storing inputted text from the user, it's safer to use `std::string`, as `std::string` will adjust automatically to hold as many characters as needed.

Modifying C-style strings

One important point to note is that C-style strings follow the same rules as C-style arrays. This means you can initialize the string upon creation, but you can not assign values to it using the assignment operator after that!

```
char str[]{ "string" }; // ok
str = "rope";           // not ok!
```

This makes using C-style strings a bit awkward.

Since C-style strings are arrays, you can use the [] operator to change individual characters in the string:

```
#include <iostream>

int main()
{
    char str[]{ "string" };
    std::cout << str << '\n';
    str[1] = 'p';
    std::cout << str << '\n';

    return 0;
}
```

This program prints:

```
string
spring
```

Getting the length of an C-style string

Because C-style strings are C-style arrays, you can use `std::size()` (or in C++20, `std::ssize()`) to get the length of the string as an array. There are two caveats here:

1. This doesn't work on decayed strings.
2. Returns the actual length of the C-style array, not the length of the string.

```
#include <iostream>

int main()
{
    char str[255]{ "string" }; // 6 characters + null terminator
    std::cout << "length = " << std::size(str) << '\n'; // prints length = 255

    char *ptr { str };
    std::cout << "length = " << std::size(ptr) << '\n'; // compile error

    return 0;
}
```

An alternate solution is to use the `strlen()` function, which lives in the `<cstring>` header. `strlen()` will work on decayed arrays, and returns the length of the string being held, excluding the null terminator:

```
#include <cstring> // for std::strlen
#include <iostream>

int main()
{
    char str[255]{ "string" }; // 6 characters + null terminator
    std::cout << "length = " << std::strlen(str) << '\n'; // prints length = 6

    char *ptr { str };
    std::cout << "length = " << std::strlen(ptr) << '\n';   // prints length = 6

    return 0;
}
```

However, `std::strlen()` is slow, as it has to traverse through the whole array, counting characters until it hits the null terminator.

Other C-style string manipulating functions

Because C-style strings are the primary string type in C, the C language provides many functions for manipulating C-style strings. These functions have been inherited by C++ as part of the `<cstring>` header.

Here are a few of the most useful that you may see in older code:

- strlen() -- returns the length of a C-style string
- strcpy(), strncpy(), strcpy_s() -- overwrites one C-style string with another
- strcat(), strncat() -- Appends one C-style string to the end of another
- strcmp(), strncmp() -- Compares two C-style strings (returns `0` if equal)

Except for `strlen()`, we generally recommend avoiding these.

Avoid non-const C-style string objects

Unless you have a specific, compelling reason to use non-const C-style strings, they are best avoided, as they are awkward to work with and are prone to overruns, which will cause undefined behavior (and are potential security issues).

In the rare case that you do need to work with C-style strings or fixed buffer sizes (e.g. for memory-limited devices), we'd recommend using a well-tested 3rd party fixed-length string library designed for the purpose.

Best practice

Avoid non-const C-style string objects in favor of `std::string`.

Quiz time

Question #1

Write a function to print a C-style string character by character. Use a pointer and pointer arithmetic to step through each character of the string and print that character. Write a `main` function that tests the function with the string literal "Hello, world!".

Show Solution

Question #2

Repeat quiz #1, but this time the function should print the string backwards.

Show Solution