6.8b — C-style string symbolic constants

BY ALEX ON AUGUST 15TH, 2015 | LAST MODIFIED BY ALEX ON JULY 17TH, 2018

C-style string symbolic constants

In the lesson **6.6 -- C-style strings**, we discussed how you could create and initialize a C-style string, like this:

```
#include <iostream>

int main()
{
    char myName[] = "Alex";
    std::cout << myName;

    return 0;
}
```

C++ also supports a way to create C-style string symbolic constants using pointers:

```
#include <iostream>

int main()
{
    const char *myName = "Alex";
    std::cout << myName;

    return 0;
}
```

While these above two programs operate and produce the same results, C++ deals with the memory allocation for these slightly differently.

In the fixed array case, the program allocates memory for a fixed array of length 5, and initializes that memory with the string “Alex\0”. Because memory has been specifically allocated for the array, you’re free to alter the contents of the array. The array itself is treated as a normal local variable, so when the array goes out of scope, the memory used by the array is freed up for other uses.

In the symbolic constant case, how the compiler handles this is implementation defined. What *usually* happens is that the compiler places the string “Alex\0” into read-only memory somewhere, and then sets the pointer to point to it. Because this memory may be read-only, best practice is to make sure the string is const.

For optimization purposes, multiple string literals may be consolidated into a single value. For example:

```
const char *name1 = "Alex";
const char *name2 = "Alex";
```

These are two different string literals with the same value. The compiler may opt to combine these into a single shared string literal, with both name1 and name2 pointed at the same address. Thus, if name1 was not const, making a change to name1 could also impact name2 (which might not be expected).

Also, because strings declared this way are persisted throughout the life of the program (they have static duration rather than automatic duration like most other locally defined literals), we don’t have to worry about scoping issues. Thus, the following is okay:

```
const char* getName()
{
    return "Alex";
}
```

In the above code, getName() will return a pointer to C-style string “Alex”. This is okay since “Alex” will not go out of scope when getName() terminates, so the caller can still successfully access it.

To summarize, use a non-const char array when you need a string variable that you can modify later. Use a pointer to a const string literal when you need a read-only string literal.

Rule: Feel free to use C-style string symbolic constants if you need read-only strings in your program, but always make them const!

std::cout and char pointers

At this point, you may have noticed something interesting about the way std::cout handles pointers of different types.

Consider the following example:

```
#include <iostream>

int main()
{
    int nArray[5] = { 9, 7, 5, 3, 1 };
    char cArray[] = "Hello!";
    const char *name = "Alex";

    std::cout << nArray << '\n'; // nArray will decay to type int*
    std::cout << cArray << '\n'; // cArray will decay to type char*
    std::cout << name << '\n'; // name is already type char*

    return 0;
}
```

On the author's machine, this printed:

```
003AF738
Hello!
Alex
```

Why did the int array print an address, but the character arrays printed strings?

The answer is that std::cout makes some assumptions about your intent. If you pass it a non-char pointer, it will simply print the contents of that pointer (the address that the pointer is holding). However, if you pass it an object of type char* or const char*, it will assume you're intending to print a string. Consequently, instead of printing the pointer's value, it will print the string being pointed to instead!

While this is great 99% of the time, it can lead to unexpected results. Consider the following case:

```
#include <iostream>

int main()
{
    char c = 'Q';
    std::cout << &c;

    return 0;
}
```

In this case, the programmer is intending to print the address of variable c. However, &c has type char*, so std::cout tries to print this as a string! On the author's machine, this printed:

```
Q╠╠╠╠╜╡4;¿■A
```

Why did it do this? Well, it assumed &c (which has type char*) was a string. So it printed the 'Q', and then kept going. Next in memory was a bunch of garbage. Eventually, it ran into some memory holding a 0 value, which it interpreted as a null terminator, so it stopped. What you see may be different depending on what's in memory after variable c.

This case is somewhat unlikely to occur in real-life (as you're not likely to actually want to print memory addresses), but it is illustrative of how things work under the hood, and how programs can inadvertently go off the rails.

6.9 -- Dynamic memory allocation with new and delete