17.11 — C-style string symbolic constants

In the previous lesson (17.10 -- C-style strings), we discussed how to create and initialize C-style string objects:

```
#include <iostream>

int main()
{
    char name[]{ "Alex" }; // C-style string
    std::cout << name << '\n';

    return 0;
}
```

C++ supports two different ways to create C-style string symbolic constants:

```
#include <iostream>

int main()
{
    const char name[] { "Alex" };        // case 1: const C-style string initialized
with C-style string literal
    const char* const color{ "Orange" }; // case 2: const pointer to C-style string
literal

    std::cout << name << ' ' << color << '\n';

    return 0;
}
```

This prints:

```
Alex Orange
```

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

In case 1, “Alex” is put into (probably read-only) memory somewhere. Then the program allocates memory for a C-style array of length 5 (four explicit characters plus the null terminator), and initializes that memory with the string “Alex”. So we end up with 2 copies of “Alex” -- one in global memory somewhere, and the other owned by `name`. Since `name` is const (and will never be modified), making a copy is inefficient.

In case 2, how the compiler handles this is implementation defined. What *usually* happens is that the compiler places the string “Orange” into read-only memory somewhere, and then initializes the pointer with the address of the string.

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. Because these literals are constants, the compiler may opt to save memory by combining these into a single shared string literal, with both `name1` and `name2` pointed at the same address.

Type deduction with const C-style strings

Type deduction using a C-style string literal is fairly straightforward:

```
auto s1{ "Alex" };  // type deduced as const char*
auto* s2{ "Alex" }; // type deduced as const char*
auto& s3{ "Alex" }; // type deduced as const char(&)[5]
```

Outputting pointers and C-style strings

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 narr[]{ 9, 7, 5, 3, 1 };
    char carr[]{ "Hello!" };
    const char* ptr{ "Alex" };

    std::cout << narr << '\n'; // narr will decay to type int*
    std::cout << carr << '\n'; // carr will decay to type char*
    std::cout << ptr << '\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 print as strings?

The answer is that the output streams (e.g. `std::cout`) make 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 (an address), it will print the string being pointed to instead!

While this is desired most 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! And because `c` is not null-terminated, we get undefined behavior.

On the author's machine, this printed:

```
Q╠╠╠╠╣4;¿■A
```

Why did it do this? Well, first it assumed `&c` (which has type `char*`) was a C-style 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.

If you actually want to print the address of a char pointer, static_cast it to type `const void*`:

```
#include <iostream>

int main()
{
    const char* ptr{ "Alex" };

    std::cout << ptr << '\n';                           // print ptr as C-style
string
    std::cout << static_cast<const void*>(ptr) << '\n'; // print address held by ptr

    return 0;
}
```

Related content

We cover `void*` in lesson 19.5 -- Void pointers. You don't need to know how it works to use it here.

Favor std::string_view for C-style string symbolic constants

There is little reason to use C-style string symbolic constants in modern C++. Instead, favor `constexpr std::string_view` objects, which tend to be just as fast (if not faster) and behave more consistently.

Best practice

Avoid C-style string symbolic constants in favor of `constexpr std::string_view`.