12.9 — Pointers and const

learncpp.com/cpp-tutorial/pointers-and-const/

Consider the following code snippet:

```
int main()
{
    int x { 5 };
    int* ptr { &x }; // ptr is a normal (non-const) pointer

    int y { 6 };
    ptr = &y; // we can point at another value

    *ptr = 7; // we can change the value at the address being held

    return 0;
}
```

With normal (non-const) pointers, we can change both what the pointer is pointing at (by assigning the pointer a new address to hold) or change the value at the address being held (by assigning a new value to the dereferenced pointer).

However, what happens if the value we want to point at is const?

```
int main()
{
    const int x { 5 }; // x is now const
    int* ptr { &x };   // compile error: cannot convert from const int* to int*

    return 0;
}
```

The above snippet won't compile -- we can't set a normal pointer to point at a const variable. This makes sense: a const variable is one whose value cannot be changed. Allowing the programmer to set a non-const pointer to a const value would allow the programmer to dereference the pointer and change the value. That would violate the const-ness of the variable.

Pointer to const value

A **pointer to a const value** (sometimes called a `pointer to const` for short) is a (non-const) pointer that points to a constant value.

To declare a pointer to a const value, use the `const` keyword before the pointer's data type:

```
int main()
{
    const int x{ 5 };
    const int* ptr { &x }; // okay: ptr is pointing to a "const int"

    *ptr = 6; // not allowed: we can't change a const value

    return 0;
}
```

In the above example, `ptr` points to a `const int`. Because the data type being pointed to is const, the value being pointed to can't be changed.

However, because a pointer to const is not const itself (it just points to a const value), we can change what the pointer is pointing at by assigning the pointer a new address:

```
int main()
{
    const int x{ 5 };
    const int* ptr { &x }; // ptr points to const int x

    const int y{ 6 };
    ptr = &y; // okay: ptr now points at const int y

    return 0;
}
```

Just like a reference to const, a pointer to const can point to non-const variables too. A pointer to const treats the value being pointed to as constant, regardless of whether the object at that address was initially defined as const or not:

```
int main()
{
    int x{ 5 }; // non-const
    const int* ptr { &x }; // ptr points to a "const int"

    *ptr = 6;  // not allowed: ptr points to a "const int" so we can't change the
value through ptr
    x = 6; // allowed: the value is still non-const when accessed through non-const
identifier x

    return 0;
}
```

Const pointers

We can also make a pointer itself constant. A **const pointer** is a pointer whose address can not be changed after initialization.

To declare a const pointer, use the `const` keyword after the asterisk in the pointer declaration:

```
int main()
{
    int x{ 5 };
    int* const ptr { &x }; // const after the asterisk means this is a const pointer

    return 0;
}
```

In the above case, `ptr` is a const pointer to a (non-const) int value.

Just like a normal const variable, a const pointer must be initialized upon definition, and this value can't be changed via assignment:

```
int main()
{
    int x{ 5 };
    int y{ 6 };

    int* const ptr { &x }; // okay: the const pointer is initialized to the address
of x
    ptr = &y; // error: once initialized, a const pointer can not be changed.

    return 0;
}
```

However, because the *value* being pointed to is non-const, it is possible to change the value being pointed to via dereferencing the const pointer:

```
int main()
{
    int x{ 5 };
    int* const ptr { &x }; // ptr will always point to x

    *ptr = 6; // okay: the value being pointed to is non-const

    return 0;
}
```

Const pointer to a const value

Finally, it is possible to declare a **const pointer to a const value** by using the `const` keyword both before the type and after the asterisk:

```
int main()
{
    int value { 5 };
    const int* const ptr { &value }; // a const pointer to a const value

    return 0;
}
```

A const pointer to a const value can not have its address changed, nor can the value it is pointing to be changed through the pointer. It can only be dereferenced to get the value it is pointing at.

Pointer and const recap

To summarize, you only need to remember 4 rules, and they are pretty logical:

- A non-const pointer can be assigned another address to change what it is pointing at.
- A const pointer always points to the same address, and this address can not be changed.

- A pointer to a non-const value can change the value it is pointing to. These can not point to a const value.
- A pointer to a const value treats the value as const when accessed through the pointer, and thus can not change the value it is pointing to. These can be pointed to const or non-const l-values (but not r-values, which don't have an address).

Keeping the declaration syntax straight can be a bit challenging:

- A `const` before the asterisk is associated with the type being pointed to. Therefore, this is a pointer to a const value, and the value cannot be modified through the pointer.
- A `const` after the asterisk is associated with the pointer itself. Therefore, this pointer cannot be assigned a new address.

```
int main()
{
    int v{ 5 };

    int* ptr0 { &v };             // points to an "int" but is not const itself, so
this is a normal pointer.
    const int* ptr1 { &v };       // points to a "const int" but is not const itself,
so this is a pointer to a const value.
    int* const ptr2 { &v };       // points to an "int" and is const itself, so this
is a const pointer (to a non-const value).
    const int* const ptr3 { &v }; // points to a "const int" and is const itself, so
this is a const pointer to a const value.

    // if the const is on the left side of the *, the const belongs to the value
    // if the const is on the right side of the *, the const belongs to the pointer

    return 0;
}
```