Const

const

- The keyword const (short for constant) is a modifier used to enforce that the variable value cannot change:
  - “change” can mean different things
- It is a modifier you can put on most any type

The big ideas

- Two kinds
  - **top level**: Locks the memory location of the variable so that its value cannot be changed.
  - **low level**: A “gateway” (pointer or ref). Through this gateway you cannot change a particular memory location

Example 5.3

Must init a const

- It is probably obvious that you must initialize a `const` variable in the declaration
  - You can’t change it once you make it, so you must init it at declare time

const does not follow copy

- `my_long = c_long;`
- Assignment is a **copy operation** (but of course there are exceptions)
- I can copy a value from a constant into another variable.
  - No restrictions there.
- Top-level locks a **memory location**, low-level a door to a location.
  - Copy is fine.

Low-level, ref/ptr

- If you want to make a variable a `const` value, then a reference or pointer to a `const` value must also be `const`
  - These types can modify the value so to prevent that they must be `const`
  - The compiler (not anything in the runtime) enforces this

You cannot remove const

- Once you make a value `const`, you cannot change it (cannot cast it away)
  - Well, not exactly. There is a `const_cast`, similar to `static_cast`, which casts away `const`-ness, but with restrictions

You can add const

- You can add `const` to a ref/ptr to a non-`const` value
  - The result is that even though the value can be changed it cannot be changed **through this ref/ptr**
  - This turns out to be very useful in functions a bit later on

const ptr

- There are really two things you might make const in a pointer
  - Its top-level: what it points to
  - Its low-level: points to a const location
- Since this is C++, we can do both

```
const long * ptr_c_long = &c_long;
```

- A pointer that can point to a `const` value. This is low level.
- `const` is in front of the type. You can change what the pointer points to but this pointer can point to constant things.

```
long* const c_p_long = &my_long;
```

- The `const` above appears **after** the original type (to the right of the long). This `const` refers to the memory address the pointer points to. This is top level
- You cannot change what the pointer points to (cannot point to a different address), but **can** change value there

```
const long* const c_c_p_long = &c_long;
```

- Do it all on one line. Easiest to read from right to left
  - Constant pointer
  - To a long
  - In fact, a constant long
- Can’t change the pointer nor the value there either.

Still Confused?

https://stackoverflow.com/questions/1143262/what-is-the-difference-between-const-int-const-int-const-and-int-const

https://www.learncpp.com/cpp-tutorial/610-pointers-and-const/

Which of these statements are FALSE about references and pointers?

- It is not possible to refer directly to a reference object after it is defined; any occurrence of its name refers directly to the object it references.
- Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.
- References cannot be null, whereas pointers can; every reference refers to some object, although it may or may not be valid. Note that for this reason, containers of references are not allowed.
- You just copied the above from https://en.wikipedia.org/wiki/Reference_(C%2B%2B)

Type Inference

C++ to the rescue

- Ok, maybe not rescue but a little help anyway
- Example 5.4

Types are a pain

- We are spending time on types because
  - C++ is crazy about types
  - The whole C++ system depends on getting things right at the type level
- C++11 people knew that and threw us some bones to make it a little easier

A using alias

- `using clc_ptr = const long* const;`
- `clc_ptr` is now a **type** (one that you have defined) and it can be used anywhere a type is needed
- `clc_ptr ptr = &my_long;`

typedef

- `typedef` is the old way (if you’ve done some C++).
- The using alias has some advantages in templates (later)
- Very little reason to use typedef any more.

auto

- The `auto` keyword has the following, very explicit, meaning. Be careful that you follow it.
- If the compiler **at compile time** can figure out in context what a type is (because it is obvious), you can declare it as type `auto`. The compiler will figure out the type and use that.

Be Clear

- Anything you `auto` **will have a type**. It is the type a variable must have to make the declaration legal
  - Ambiguous type, can't `auto` it
- You must be able to read the code and know that type as well, but it is not always obvious

Auto drops refs and const

- When it deduces types, `auto` ignores references and `const` qualifiers
- Only the base type comes through

decltype

- `decltype` is another way to auto a variable (or anything) that preserves things like const
- We’ll see more of it later.

The Unsigned Type

Integers 0 to Max

- There are a number of integer types. If such an integer is proceeded with the modifier unsigned it has the following effect
  - The integer cannot store a negative number
  - Its range is doubled

Doubled Range

- Assume 4 bytes (32 bits) for an integer
- Likely an `int` but you have to check
  - `int` $\pm 2^{31}$signed
    - Range is -2147483648 to $+2147483647$
    - Why the extra negative number?
  - `unsigned int`, $2^{32} - 1$ so 0 to 4294967295

Overflow / underflow unsigned

- C++ **guarantees** that for an unsigned value an overflow/underflow wraps to the next element in the range

```
unsigned int max_ui = pow(2, 32) - 1;
unsigned int min_ui = 0;
cout << max_ui; // 4,294,967,295
cout << max_ui + 1; // 0
cout << min_ui; // 0
cout << min_ui - 1 // 4,294,967,295
```

No guarantees on signed

- The C++ standard makes **no guarantee** on the behavior of signed overflow/underflow though it is often implemented the same

```
int max_i = pow(2, 31) - 1;
int min_i = -pow(2, 31);
cout << max_i + 1; // -2,147,483,648
cout << min_i - 1; // 2,147,483,647
```

Mixed Types

- When mixing signed and unsigned types, the compiler promotes the signed to an unsigned!

```
unsigned int max_ui = pow(2,32) - 1;
int one = 1;
cout << max_ui + one; // 0, wraps
```

All ops are converted to ints

- A short is 2 bytes (16 bits). Watch this!

```
unsigned short max_us = pow(2, 16) - 1;
unsigned short s_one = 1;
cout << max_us + s_one // 65,535!
unsigned temp = max_us + s_one;
cout << temp; // 0
```

unsigned

- The unsigned modifier is only for integer types (doesn’t make sense for floats)
  - Doubles the range a long can hold
  - Only allows values 0 or greater
    - Well “allows” is a strong word
    - The compiler will allow it
    - The result is not what you expect

When do you use unsigned?

- Somewhat controversial.
  - Some recommend never
  - Others say the guaranteed behavior is useful because overflow and underflow happen in ints as well
- Bottom line: when you absolutely know that values won’t be negative or overflow, you still likely should avoid unsigned
  - Google Style Guide:
  - "avoid unsigned types (except for representing bitfields or modular arithmetic). Do not use an unsigned type merely to assert that a variable is non-negative."

Example 5.5