Type Modifiers and References
What is a type modifier?

- C++ provides a set of modifiers that can be applied to some/all of the types in the system
  - Some are numeric specific
  - Some control variable access
  - Some change the meaning of a variable
Compiler tracks four things about variables (so it can turn stuff into assembly code)

1. Name (names, aliases), like variable names
2. Address (where it goes in memory)
3. Type (which determines how many bytes it might occupy)
4. Value
When someone says you are running a “64-bit” os/cpu/something

- The number of bits that can be used in an address (to memory) is 64
- The range of addresses (unsigned) is about $2^{64}$, 0 to $1.85 \times 10^{19}$ bytes
- That is $\sim 16$ exabytes ($10^{18}$), (1000 petabytes, 1 million terabytes, 1 billion gigabytes)
64 Bits

1100 1011 0110 1111 0000 1010 0010 0111 1100 1011 0110 1111 0000 1010 0010 0111

- Each bit represents a signal on a line
- 64 such lines going to the CPU that it can use to select a byte
- Can select one of 18 exabytes, can move 8 bytes at once
A 64-bit address looks like 0xcb6f0a27cb6f0a27

0x prefix indicates hex in C++
Hey, wait a minute?

- You said an address on a 64-bit machine was 64 bits, 16 hex numbers
  - 0xcb6f0a276f0a27
- But the address on your future examples are only 12 hex numbers (48 bits)
  - 0x7fff519b7a8c
Hardware manufacturers know (or at least surmise) that no one will have that much memory anytime soon.

Thus they cheat and provide fewer address lines since they know they won’t get used.

Saves money!
## A symbol table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_long</td>
<td>long</td>
<td>0x7fff519b7a8c</td>
<td>123</td>
</tr>
<tr>
<td>p_long</td>
<td>long*</td>
<td>0x7fff519b7a80</td>
<td>0x7fff519b7a8c</td>
</tr>
<tr>
<td>r_long</td>
<td>long&amp;</td>
<td>0x7fff519b7a8c</td>
<td>123</td>
</tr>
</tbody>
</table>
C++ Name rules

- Only alpha, digit, and underscore
- Cannot start with a digit
- Don’t use a keyword as a name
- Names are case-sensitive
  - Upper and lower-case are different
- No special characters
Which of the following are valid names?

- 42_or_fight
- sHiT
- _illegal_
- I don't know
The &

- Is a type modifier in the context of declaration (it has other meanings)
  - In a declaration, the & means a reference to another type
  - Both parts matter, the reference and the type it references
A reference is a variable declaration that is a name alias for another variable.

- It is indicated by the & (ampersand).
  - But it has different meanings in other contexts!

- It requires initialization.
  - When you declare a reference you have to say what it refers to.
  - Must be an lvalue.
    - No Literals
    - No expression results
A reference is not an object

- A reference is a name alias in the symbol table
- It does not create a new variable
- No new memory allocation
- It simply refers to an existing variable
Things to note

- Stuff happens sequentially, so if you have a variable declared before a reference, the reference can refer to it.
- In a multiple declaration, the & goes with the variable.
Which of the following are legal initializations of a reference?

- `int & x;`
- `int & x = 4;`
- `int & x = 3 + 4;`
- `None of the above`
Which of the following are legal initializations of a reference?

- `int & x = func_call();`
- `int & x = x;`
- `int & x = y;`
- None of the above
Pointers
**A symbol table**

- my_long = 123;

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_long</td>
<td>long</td>
<td>0x7fff519b7a8c</td>
<td>123</td>
</tr>
<tr>
<td>ref_long</td>
<td>long&amp;</td>
<td>0x7fff519b7a8c</td>
<td>123</td>
</tr>
</tbody>
</table>

- ref_long = 456;

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_long</td>
<td>long</td>
<td>0x7fff519b7a8c</td>
<td>456</td>
</tr>
<tr>
<td>ref_long</td>
<td>long&amp;</td>
<td>0x7fff519b7a8c</td>
<td>456</td>
</tr>
</tbody>
</table>
Pointers, an address type

- A pointer is a variable whose value is an *address*
  - It has a value, but the value is to *another location in memory*
  - As a result, a pointer can “point to” another variable
  - Can refer to another variable in memory by that other variable’s memory address
• Pointers are a topic much discussed in CS
  • Python and Java don’t have them (sort of) because they can be the source of so many problems
  • Tend to be confusing to beginner programmers
  • Is really a pretty easy to understand subject as long as you are careful
In the context of a declaration, a star (*) following the type means that the variable being declared is a pointer.

```c
long* my_pointer; // pointer to long
```
Like &, * follows the variable

- Like we saw in &, the * goes with the variable, not the type
- This is unfortunate. We’d like to say the type is long*, but the * only applies to the next var:

```c
long* p_long, my_long; // Confusing
long *p_long, my_long; // Less confusing
```
The *

- * is a type modifier that means the type is a pointer to some other type
- Both matter: a pointer and to some type
Example 5.2
What is the size of a pointer?

- Another question, what kind OS/CPU is this?
  - If 32-bit then every pointer is 4 bytes
  - If 64-bit then every pointer is 8 bytes
- Why?
Could be 6 bytes, but...

- Since addresses are actually 48 bits, they could fit in 6 bytes, but the hardware is setup to fetch 8 bytes at a time (that is the data lines are in fact 64 bits wide) and so they do.
- Might as well use 8 bytes. Perhaps someday memory will catch up.
In the context of an expression, as a unary operator, the * represents “dereference”.

The pointer has an address as its value. Dereferencing means to use the value that the pointer has as its value to either fetch or set a value.
Dereferencing, lvalue vs rvalue

- This is kind of intuitive, but we need to be clear
- Dereferencing as an rvalue provides a value at the address pointed to
  \[ x = *y; \]
- Dereferencing as an lvalue provides a memory location where values can be stored
  \[ *x = y; \]

\[ *x = *y; \]
Another meaning for &

- In an expression, the & means “address of”
  - These are the kinds of values stored in a pointer
Empty Pointer

- In the previous code, the pointer p_long points to address 0 (nothing)
  - It is an object
  - It has an address
  - Its value is indeterminate, maybe 0x0?
- Dereferencing a pointer to 0x0 is illegal. It compiles but fails at runtime
## Before setting `p_long`

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_long</td>
<td>long</td>
<td>0x7fff519b7a8c</td>
<td>123</td>
</tr>
<tr>
<td>p_long</td>
<td>long*</td>
<td>0x7ff519b7a80</td>
<td>0</td>
</tr>
</tbody>
</table>

What is 0? What does that point to?
After setting `p_long = &my_long`

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_long</td>
<td>long</td>
<td>0x7fff519b7a8c</td>
<td>123</td>
</tr>
<tr>
<td>p_long</td>
<td>long*</td>
<td>0x7ff519b7a80</td>
<td>0x7fff519b7a8c</td>
</tr>
</tbody>
</table>

Value of `p_long` is the address of `my_long`
\[ *p\_long = 456 \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_long</td>
<td>long</td>
<td>0x7ff519b7a8c</td>
<td>456</td>
</tr>
<tr>
<td>p_long</td>
<td>long*</td>
<td>0x7ff519b7a80</td>
<td>0x7ff519b7a8c</td>
</tr>
</tbody>
</table>

3 step process:
1. Get the value of p\_long
2. p\_long value is an address, go there
3. Set the value of that address to the new value
long &r_long = *p_long

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_long</td>
<td>long</td>
<td>0x7fff519b7a8c</td>
<td>456</td>
</tr>
<tr>
<td>p_long</td>
<td>long*</td>
<td>0x7ff519b7a80</td>
<td>0x7fff519b7a8c</td>
</tr>
<tr>
<td>r_long</td>
<td>long&amp;</td>
<td>0x7fff519b7a8c</td>
<td>456</td>
</tr>
</tbody>
</table>

3 step process:
1. Get the value of p_long
2. p_long value is an address, go there
3. Set the value of that address to the new value
Though it seems easy enough, pointers tend to be a hard topic.

- Hard to do correctly
- Introducing early to get the hang of it as we go