Type Modifiers

James Daly
CSE 232
Michigan State University
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
Tracking

- Compiler tracks four things (so it can turn stuff into assembler)
  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 \(~16\) exabytes ($10^{18}$), (1000 petabytes, 1 million terabytes, 1 billion gigabytes)
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
Binary = Decimal = Hex

- 0000 = 0
- 0001 = 1
- 0010 = 2
- 0011 = 3
- 0100 = 4
- 0101 = 5
- 0110 = 6
- 0111 = 7
- 1000 = 8
- 1001 = 9
- 1010 = 10 = a
- 1011 = 11 = b
- 1100 = 12 = c
- 1101 = 13 = d
- 1110 = 14 = e
- 1111 = 15 = f
64 Bits Addresses

1100 1011 0110 1111 0000 1010 0010 0111 1100 1011 0110 1111 0000 1010 0010 0111
  c   b  6   f   0   a  2   7   c   b  6   f   0   a  2   7

- 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 address on your examples are only 12 hex numbers (48 bits)
  - 0x7fff519b7a8c
Expediency

- 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>0</td>
</tr>
<tr>
<td>r_long</td>
<td>long&amp;</td>
<td>0x7fff519b7a8c</td>
<td>123</td>
</tr>
</tbody>
</table>
Names

- Example 5.1
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
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
References, a name alias

- 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.
Example 5.1
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>
Address

- Example 5.2
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
A word on pointers

- 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.
Dereferencing

- 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
- Dereferencing as an lvalue provides a memory location where values can be stored
Another meaning for &

- In an expression, the & means “address of”
  - These are the kinds of values stored in a pointer
Example 5.2
Empty Pointer

- In the before section, the pointer p_long points to 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`
Example 5.2
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>0x7fff519b7a8c</td>
<td>456</td>
</tr>
<tr>
<td>p_long</td>
<td>long*</td>
<td>0x7fff519b7a80</td>
<td>0x7fff519b7a8c</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
Values

- Example 5.3
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` 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.
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)
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 as well, not always obvious
Auto drops refs and const

- When it deduces types, `auto` ignores references and `const` qualifiers
- Only the 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.
Unsigned

- Example 5.5
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 ±2^{31}` signed
    - Range is $-2^{147483649}$ to $+2^{147483647}$
    - 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

```cpp
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

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

```c
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!

```cpp
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, unsigned is fine
Example 5.5