COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

Type Modifiers

COMPUTER SCIENCE DEPARTMENT 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

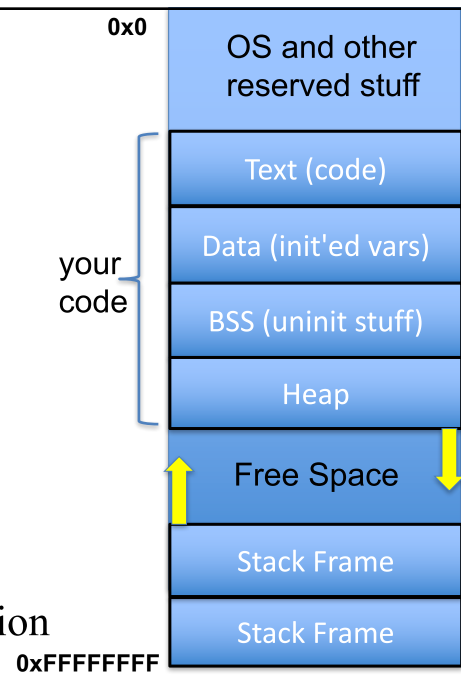
Type Modifiers

Tracking

Compiler tracks four things (so it can turn stuff into assembler)

1. name (names, aliases), think var name
2. address (where it goes in memory)
3. its type (which means how many bytes it might occupy)
4. its value

- lower at the top
- higher at the bottom
- **text section** has your code
- **data section** has initialized variables
- **bss section** has uninitialized memory
- **heap** grows down, dynamic memory
- **stack** grows up, function calls



Let's remember

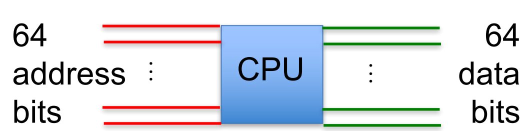
When someone says you are running a "64 bit" os/cpu/something, remember what that means:

- 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 $1.85*10^{19}$ bytes
- that is ~16 exabytes(10^{18}), (1000 petabytes, 1 millon 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

Binary = Decimal = Hex

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

64 bit 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:

0xcb6f0a27cb6f0a27

But addresses 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 the "cheat" and provide fewer address lines since they won't likely get used. Saves money!

A symbol table

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	123
p_long	long*	0x7fff519b7a80	0
r_long	long&	0x7fff519b7a8c	123

Names

Example 5.1

C++ Name Rules

1. Only alpha, digit and underscore
2. Cannot start with a digit
3. Don't use a keyword as a name
4. Names are case-sensitive
   1. upper and lower case are different
5. No special characters.

The &

Is a type modifier, in the context of a 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, context!
- it **requires** initialization
  - when you declare a reference, you have to say what it refers to
  - must be an lvalue, so no literals or expression results.

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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 simple refers to an existing variable.

Type Modifiers

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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

Type Modifiers

Ex 5.1

```
int main(){
  long my_long = 27, a_long=56;
  long &ref_long = my_long; // & in decl, a ref
  // one ref, one long (& goes with var)
  long &ref2_long = a_long, last_long = 123;
  //long &ref_long2 = 27;  // ERROR,no rvalues
  cout << "Long:"<<my_long<<", Ref:"
       <<ref_long<<endl;
  my_long = 123;  // alias, ref_long changes
  cout << "Long:"<<my_long<<", Ref:"
       <<ref_long<<endl;
  ref_long = 456; // ditto
  cout << "Long:"<<my_long<<", Ref:"
       <<ref_long<<endl;
}
```

Type Modifiers

A symbol table

```
my_long = 123;
```

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	123
ref_long	long&	0x7fff519b7a8c	123

```
ref_long = 456;
```

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	456
ref_long	long&	0x7fff519b7a8c	456

Type Modifiers

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

A word on pointers

Pointers are a topic much discussed in CS.

- Python and Java don't have them, because they can be the source of so many problems
- Tend to be confusing to beginning programmers
- is really a pretty easy to understand subject, as long as you are careful

* for pointer

In the context of a declaration, a star (*) following the type means that the variable being declared is a pointer.

```
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 that the type is int*, but the * only applies to the next var:

```
long* p_long, my_long; //type clear, confusing
long  *p_long, my_long; // less confusing
```

The *

* is a type modifier that means that the type is a pointer to some other type

- both matter. A pointer and to some type

Ex 5.2

```
int main (){
  long my_long = 123;
  long *p_long, a_long; // * means pointer, a_long just an long
  double my_double = 3.14159, *p_double;

  cout << "Size of long ptr:"<<sizeof(p_long)<<endl;
  cout << "Size of double ptr:"<<sizeof(p_double)<<endl;

  // & is "address of"
  cout << "Before setting pointer value"<<endl;
  cout << "Addr of long:"<<&my_long
       <<", Val of long:"<<my_long<<endl;
  cout << "Addr of ptr:"<<&p_long
       <<", Val of ptr:"<<p_long<<endl;
  p_long = &my_long;
  cout << "After setting pointer value"<<endl;
  cout << "Addr of long:"<<&my_long
       <<", Val of long:"<<my_long<<endl;
  cout << "Addr of ptr:"<<&p_long<<", Val of ptr:"<<p_long<<endl;
…
```

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, `long` or `double` 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.

A dereferencing as an rvalue provides a value at the address pointed to

A 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.

Ex 5.2

```
int main (){

  long my_long = 123;
  long *p_long, a_long; // * means pointer, a_long just an long
  double my_double = 3.14159, *p_double;

  cout << "Size of long ptr:"<<sizeof(p_long)<<endl;
  cout << "Size of double ptr:"<<sizeof(p_double)<<endl;

  // & is "address of"
  cout << "Before setting pointer value"<<endl;
  cout << "Addr of long:"<<&my_long
       <<", Val of long:"<<my_long<<endl;
  cout << "Addr of ptr:"<<&p_long
       <<", Val of ptr:"<<p_long<<endl;
  p_long = &my_long;
  cout << "After setting pointer value"<<endl;
  cout << "Addr of long:"<<&my_long
       <<", Val of long:"<<my_long<<endl;
  cout << "Addr of ptr:"<<&p_long<<", Val of ptr:"<<p_long<<endl;
…
```

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?

Deferencing a pointer to 0x0 is illegal. It compiles, but fails at run

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

before setting p_long

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	123
p_long	long*	0x7fff519b7a80	0

Value is 0, what does that point to????

Type Modifiers

Ex 5.2

```
int main (){

  long my_long = 123;
  long *p_long, a_long; // * means pointer, a_long just an long
  double my_double = 3.14159, *p_double;

  cout << "Size of long ptr:"<<sizeof(p_long)<<endl;
  cout << "Size of double ptr:"<<sizeof(p_double)<<endl;

  // & is "address of"
  cout << "Before setting pointer value"<<endl;
  cout << "Addr of long:"<<&my_long
       <<", Val of long:"<<my_long<<endl;
  cout << "Addr of ptr:"<<&p_long
       <<", Val of ptr:"<<p_long<<endl;
  p_long = &my_long;
  cout << "After setting pointer value"<<endl;
  cout << "Addr of long:"<<&my_long
       <<", Val of long:"<<my_long<<endl;
  cout << "Addr of ptr:"<<&p_long<<", Val of ptr:"<<p_long<<endl;
…
```


p_long = &my_long;

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	123
p_long	long*	0x7fff519b7a80	0x7fff519b7a8c

value of p_long is the address of my_long

```
long my_long = 123;
long* p_long = nullptr;
```

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	123
p_long	long*	0x7fff519b7a80	0

```
long* p_long = &my_long
```

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	123
p_long	long*	0x7fff519b7a80	0x7fff519b7a8c


```
  long my_long = 123;
  long *p_long;
  p_long = &my_long;
…
  // * is "points to"
  cout << "Val of ptr:"<<p_long<<", ptr points to:"<<*p_long<<endl;
  *p_long = 456;      // change the value which p_long "points to"
  cout << "Val of long:"<<my_long<<", val of ptr:"<<p_long<<endl;

  // now a reference
  // p_long is obj, so is what it points to. So OK
  long &r_long = *p_long;
  cout << "Addr of long:"<<&my_long
                       <<", Val of long:"<<my_long<<endl;
  cout << "Addr of ptr:"<<&p_long<<", Val of ptr:"<<p_long<<endl;
  cout << "Addr of ref:"<<&r_long<<", Val of ref:"<<r_long<<endl;

}
```

*p_long = 456

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	456
p_long	long*	0x7fff519b7a80	0x7fff519b7a8c

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 my_long = 123;
  long *p_long;
  p_long = &my_long;
…
  // * is "points to"
  cout << "Val of ptr:"<<p_long<<", ptr points to:"<<*p_long<<endl;
  *p_long = 456;      // change the value which p_long "points to"
  cout << "Val of long:"<<my_long<<", val of ptr:"<<p_long<<endl;

  // now a reference
  // p_long is lvalue, so is what it points to. So OK
  long &r_long = *p_long;
  cout << "Addr of long:"<<&my_long
                        <<", Val of long:"<<my_long<<endl;
  cout << "Addr of ptr:"<<&p_long<<", Val of ptr:"<<p_long<<endl;
  cout << "Addr of ref:"<<&r_long<<", Val of ref:"<<r_long<<endl;
}
```

long &r_long = *p_long;

Name	Type	Address	Value
my_long	long	0x7fff519b7a8c	456
p_long	long*	0x7fff519b7a80	0x7fff519b7a8c
r_long	long&	0x7fff519b7a8c	456

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

Hard topic

Though it seems easy enough, pointers tend to be a hard topic.

- hard to do correctly
- introducing early, get the hang of it as we go.

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

Values

Example 5.3

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

const

The keyword `const` (short for constant of course) 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.

Type Modifiers

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.

Ex 5.3

```
int main(){
  long my_long = 10, a_long = 20;
  const long c_long = 123;  // constant long must be inited, cannot change
  // const long x;              // ERROR, must init
  // c_long = 456;          // ERROR, can't change a const
  my_long = c_long; // assign is copy, orig not changed. So OK

  // references
  const long &ref1_long = c_long; // ref cannot change referenced value
  const long &ref2_long = my_long;    // can ref a non-const, ref still can't change
  const double &ref_pi = 3.14159;  // can even const ref a literal
  // ref2_long = 100;               // ERROR, cannot change since ref is const
                                 // even though what it refs is non-const
  a_long = ref1_long;   // assign is copy, orig not changed. So OK
```

Must init a const

It is probably obvious that you must initalize 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

Following up on the same idea, if you want to make a variable a `const` value, then the 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 in fact, similar to a `static_cast`, a `const_cast` which casts away const-ness, but with restrictions!

you can add const

You can add `const` to a var/ref/ptr to a non-const value:

- the result is that even though the value can be changed, it cannot be changed ***through this var/ref/ptr***
- that turns out to be very useful in functions a bit later on

Ex 5.3

```
// references
 const long &ref1_long = c_long; // ref cannot change referenced value
 const long &ref2_long = my_long;    // can ref a non-const, ref still can't change
 const double &ref_pi = 3.14159;  // can even const ref a literal
 // ref2_long = 100;               // ERROR, cannot change since ref is const
                                  // even though what it refs is non-const
 a_long = ref1_long;   // assign is copy, orig not changed. So OK

 // pointers
 const long *ptr_c_long = &c_long; // low level, ptr to const long
 ptr_c_long = &a_long;               // can point to a non-const
 // *ptr_c_long = 27;                // ERROR, can't change through const ptr
 long *const const_ptr_my_long = &my_long; // top level, constant ptr
 // const_ptr_my_long = &a_long;         // ERROR, cannot change what is pointed to
 const long * const c_c_p_long = &c_long;
}
```

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.

So since this is C++, we can do both

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

`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.

Type Modifiers

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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.

Type Modifiers

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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 an long
- in fact a constant long

Can't change the pointer, can't change the value there either.

Type Modifiers

C++11 to the rescue

OK, not rescue but a little help anyway

Ex 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).

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 it out the type and use that.

Be clear

Anything you `auto` *will have a type*, it is the type, the *obvious type*, that 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.

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

decltype

`decltype` is another way to auto a variable (or anything) that preserves things like const.

We'll see it more later.

Type Modifiers

unsigned

Ex 5.5

integers 0 to max

There are a number of integer types. If such an integer is preceeded with the modifier unsigned it has the following effect:

- the integer cannot (at least accurately) store negative number
- its range is doubled

doubled range

Assume 4 bytes (32 bits) for an integer. For us, likely `int` but you have to check.

- `int`, $\pm 2^{31}$ signed. Range is
- - 2'147'483'648 to 2'147'483'647
  - why the extra negative number?
- `unsigned int`, $2^{32} - 1$, so 0 - 4'294'967'295

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

overflow/underflow unsigned

C++ ***guarantees*** that for an unsigned value, on 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
```

Type Modifiers

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
```


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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
```

Type Modifiers

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
```


COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

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, but the result is not what you would think it is.

Type Modifiers

When do you use unsigned

Somewhat controversial. Google for example recommends never, others say the guaranteed behavior is useful because overflow and underflow happens in ints as well.

Bottom line: when you absolutely, positively know that values won't be negative or overflow, unsigned is fine.

Example 5.5

```
int main () {
    unsigned long my_ulong = 23;
    cout << "Unsigned long:"<<my_ulong<<endl;
    my_ulong = -23;
    //whaaaat?
    cout << "Unsigned long:"<<my_ulong<<endl;
}
```

Does this compile
If so what does second cout print?