*A story of & and **

A symbol table

4 properties of any C/C++ variable:
- name
- type
- address (in memory, probably its virtual address
- value

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

The &

As a type modifier:

- 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.

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.

Things to note

- When you declare a reference, you have to say (at that time) what it refers to
  - no such thing as an empty reference.
- in a multiple declaration, the & goes with the variable

Ex: references

```
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;
}
```


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

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

* 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: pointers

```
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??

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

Another meaning for &

In an expression, the & means "address of".

- These are the kinds of values stored in a pointer.

Ex: pointers

```
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

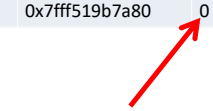
In the before section, the pointer `p_long` points to nothing:

- 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

before setting p_long

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



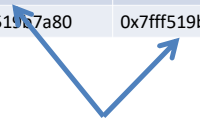
Value is 0, what does that point to????

Ex: pointers

```
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

Ex: pointers

```
  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