Pointers in C++

(Basically) everything in C++ resides somewhere in memory

```
int main() {
   int a = 23;
   cout << "a's value is " << a;
}
```

“a” has an address

But what is it?

```
int main() {
   int a = 23;
   cout << "a's value is " << a;
}
```

&a

Give me the address of a

Please

Please

Please

Don’t get this confused with a reference

They are totally different things!

Please

Please

Please

Don’t get this confused with a reference

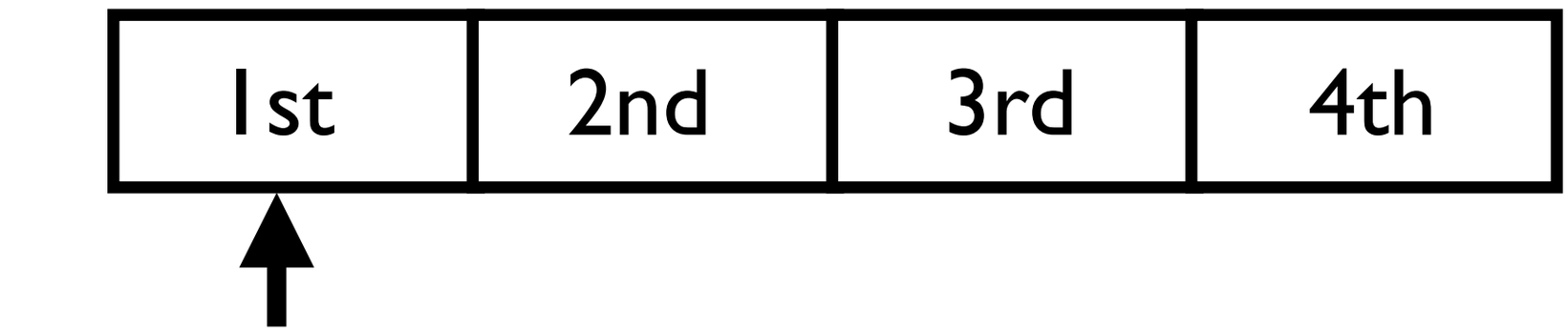
They are totally different things!

(Forget about references for now..)

Everything in C++ also has a size

sizeof(a)

This whole thing is a



Local variables are stored on the **stack**

Variables next to each other, are placed next to each other on the stack

So now, what will **this** do?

```
int main() {
  int a = 23;
  int b = 24;
  cout << "a's value is " << a << endl;
  cout << "a's address is " << &a << endl;
  cout << "the size of a is " << sizeof(a) << endl;
  cout << "b's value is " << b << endl;
  cout << "b's address is " << &b << endl;
  cout << "the size of b is " << sizeof(b) << endl;
}
```

Assuming a is 0x7fff5388990c

Lesson: the stack grows **down**

When C++ **calls** a function, it creates space for its local variables on the stack

When C++ **returns from** a function, it destroys those by moving the stack up

Let’s call another…

If I know someone's address, I can go get the data at that address…

*(&a) is the same as a

I can use the * operator to get the data at some address

I can even **store** addresses

```
int *pointerToA = &a;
```

```
void storingPtr() {
  int a = 23;
  int *pointerToA = &a;
  cout << "a's value is " << *pointerToA << endl;
  cout << "&a is " << &a << endl;
  cout << "pointerToA is " << pointerToA << endl;
  cout << "&pointerToA is " << &pointerToA << endl;
  cout << "sizeof(pointerToA) is " << sizeof(pointerToA)
       << endl;
  return;
}
```

Note: all pointers take up the same number of bytes

And that number depends on your machine (32/64-bit)

What happens if I want to use a pointer after the function returns?

```
int *returnsABadPointer() {
  int a = 23;
  int *ptr = &a;
  cout << "the value of *ptr is " << *ptr << endl;
  return &a;
}
```

```
void doSomethingBad() {
  int *ptr = returnsABadPointer();
  cout << "the value of *ptr is " << *ptr << endl;
  return;
}
```

Lesson: once the function returns, that pointer is **meaningless**

Lesson: once the function returns, that pointer is **meaningless**

Even worse, if I continue to use it, it could cause **security problems**

Lesson: once the function returns, that pointer is **meaningless**

Even worse, if I continue to use it, it could cause **security problems**

An attacker could figure out how to load their data into *ptr and control my code

So how can I hold onto things after returns!?

I use the **heap**

Unlike the stack, when I put things on the heap,
they stay there until I tell them to go away

```
int *newPointer() {
  int *a = new int;
  *a = 23;
  return a;
}
```

```
void doSomethingFine() {
  int *ptr = newPointer();
  cout << "the value of *ptr is " << *ptr;
  *ptr++
  cout << "the value of *ptr is " << *ptr;
  delete ptr;
}
```

```
int *newPointer() {
  int *a = new int;
  *a = 23;
  return a;
}
```

```
void doSomethingFine() {
  int *ptr = newPointer();
  cout << "the value of *ptr is " << *ptr;
  *ptr++
  cout << "the value of *ptr is " << *ptr;
  delete ptr;
}
```

Note: I used **delete**

If I don't remember to call **delete**, the memory will never go away

It will live on forever, like a zombie

Gradually, the world will be taken over…

```
void useAllMyMemory() {
  for (long long i = 0; i < 12346789000; i++) {
    int *ptr = new int;
  }
  return;
}
```

```
void dontUseAllMyMemory() {
  for (long long i = 0; i < 12346789000; i++) {
    int *ptr = new int;
    delete ptr;
  }
  return;
}
```


CRISIS
AVERTED
memegenerator.net