

C++: Nuts and Bolts


Logistics

This week…

C++

Logistics

Next week…

Question:
Talk to the person next to you.

- Come up with a programming language (not necessarily that you know)
- List one (nominal) advantage
- One potential (disadvantage)

- C++
- It's fast and lets you interact closely with hardware
- Tons and tons of features, not clear which ones to use when. Requires manual memory management, causes tons of potential security errors, etc…

Understanding how C++ lays out memory is **key** to mastering C++

C++’s view on memory…

At it’s core, everything in C++ is about manipulating bytes, or sequences of bytes

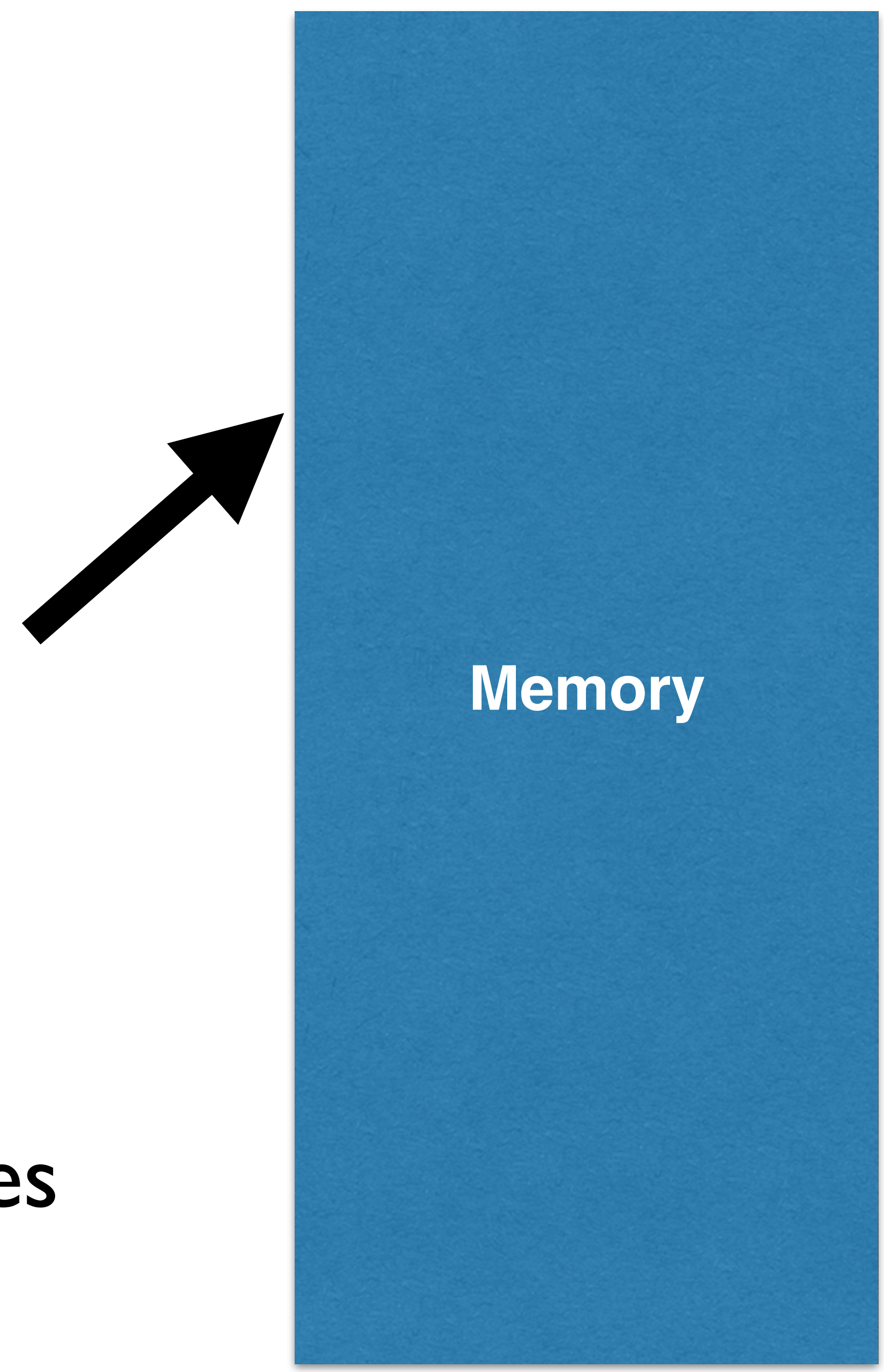
Memory Management

- Each variable in C++ exists somewhere in memory

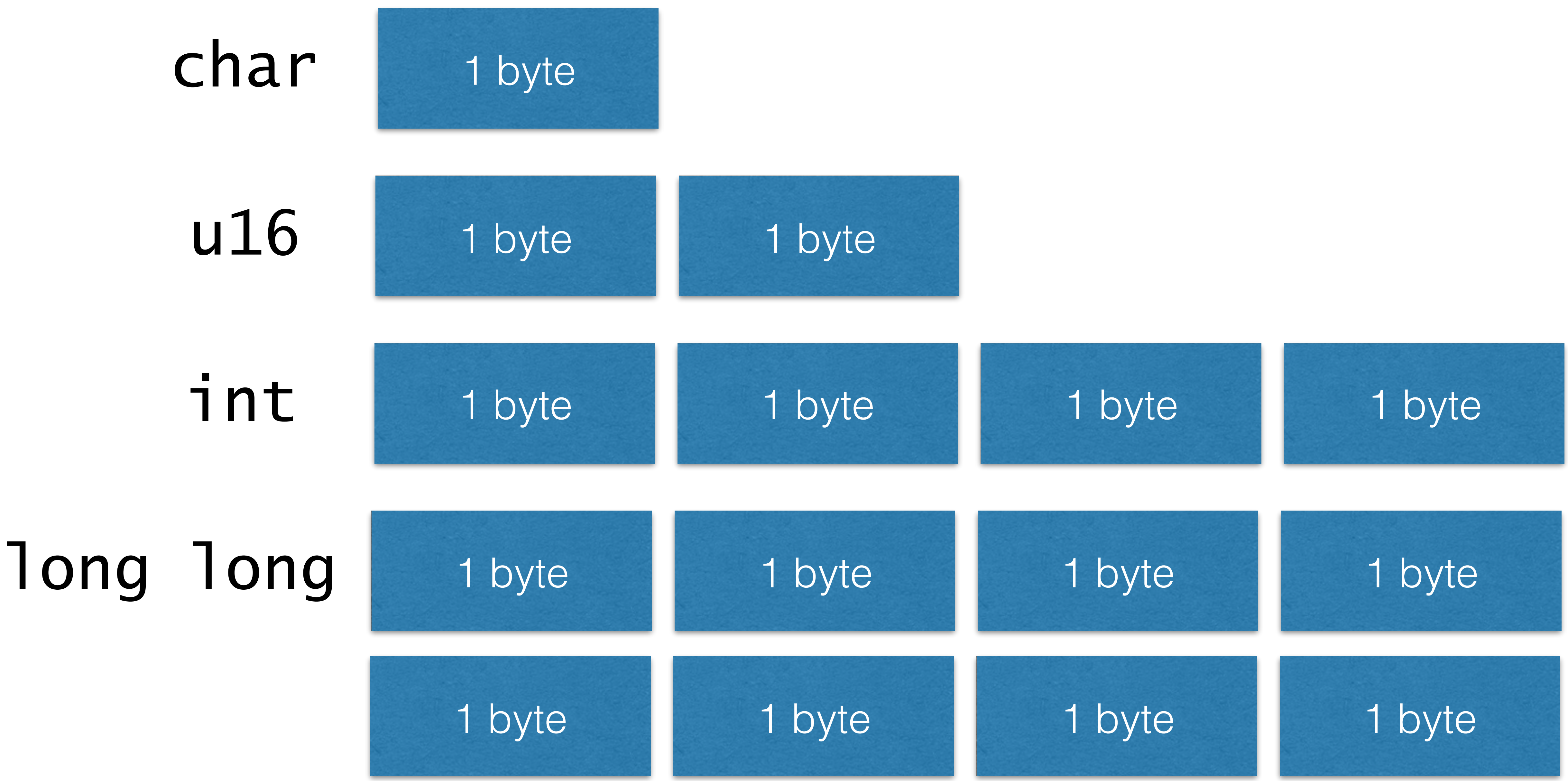
C++ thinks of this as a giant array of bytes

There are no types, everything is just a byte.

The way you **use** C++ determines what those bytes mean



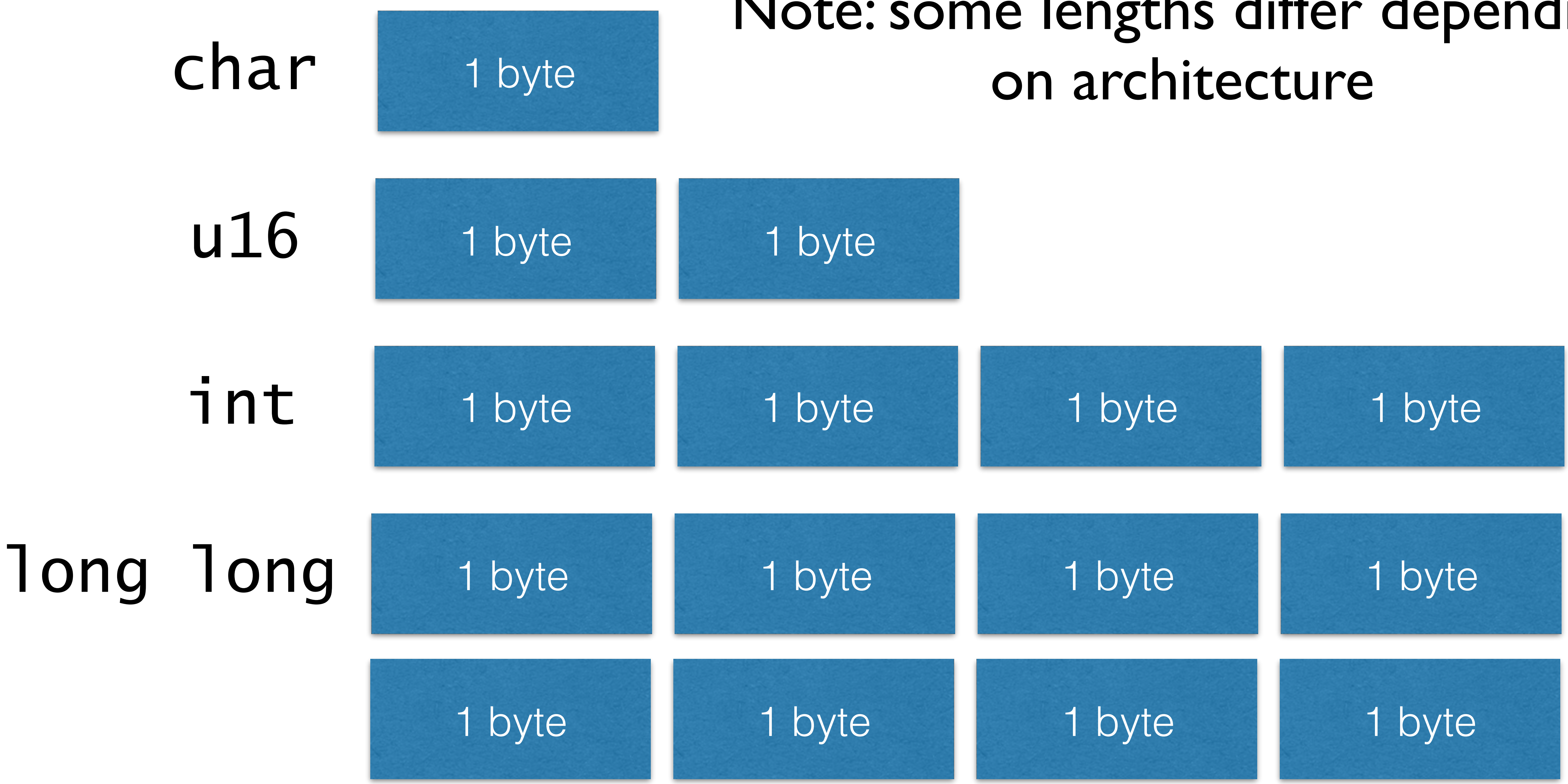
Primitive types



(for a 32-bit architecture…)

Primitive types

Note: some lengths differ depending on architecture



(for a 32-bit architecture…)

(nearly) everything in C has an **address**

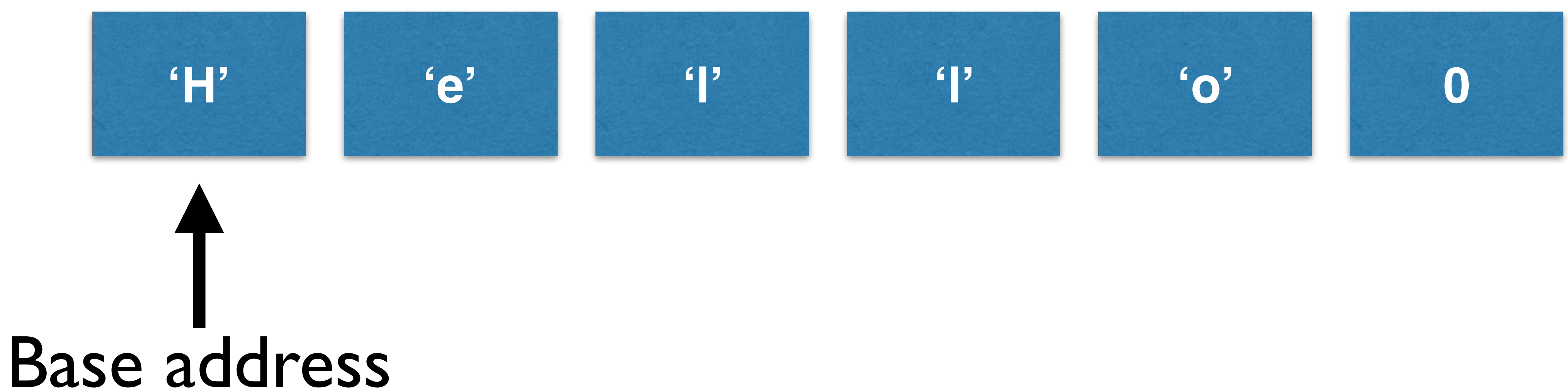
(e.g., an integer literal does not)

```
int main() {
  int x;
  cout << "The address of x is " << &x << "\n";
}
```

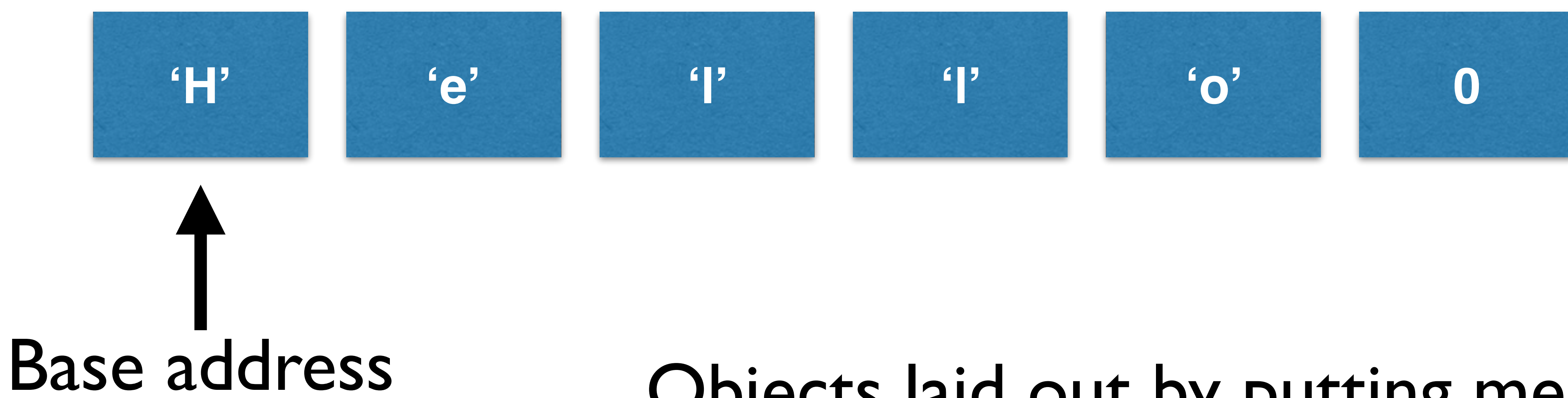
```
Kyles-MacBook-Pro-2:c++play micinski$ ./a.out
The address of x is 0x7fff5547347c
Kyles-MacBook-Pro-2:c++play micinski$
```

We will use this later to **refer** to things

What a C-style string looks like…



What a C-style string looks like…



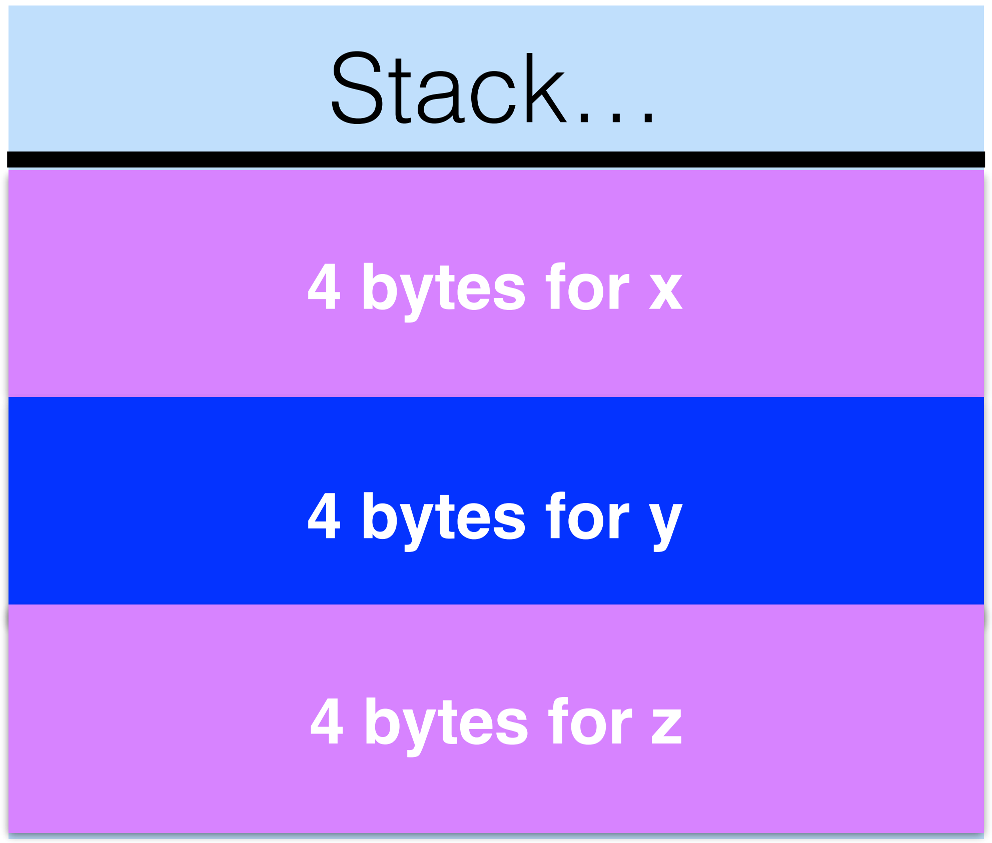
Objects laid out by putting member variables in sequence, just like strings

C++ types are used to interpret regions of memory in the right way

Stack Layout for Functions

- Local variables are laid out on the **stack**
- Each time function called, new region of memory allocated for that call:
  - Activation Record / Stack Frame

```
int sum(int x, int y) {
  int z = x + y;
  return z;
}
```



Stack Overflows

- Each time a function is called, the variables for the function are pushed onto the stack
- Typical computer has a small stack, relatively speaking
  - ~4MB
- Long seqs of recursive calls potentially overflow stack

```
int bad_sum(int x, int y) {
  if (x == 0) {
    return y;
  } else {
    return 1 + bad_sum(x - 1, y);
  }
}
```

```
Kyles-MacBook-Pro-2:c++play micinski$ g++ ex.cc; ./a.out
The address of x is 0x7fff5d69a47c
Calculating 2352322 plus 42
Segmentation fault: 11
Kyles-MacBook-Pro-2:c++play micinski$
```

This has led to a **false** assumption among C++ programmers that recursion is bad / slow

Recursion—when used and implemented correctly—is powerful tool

Tail Recursion

- A function is **tail-recursive** if recursive call is the last operation
  - Tail recursive: `return sum(x - 1, acc + 1)`
  - Not tail recursive: `return 1 + sum(x-1)`
- **Tail-recursion** can be optimized so that it doesn't use stack
- Basic intuition: stack used for **partial results**
  - If you don't do anything after recursive call, no need for that!

Code covers…

- Use of vector
- Iterating using C++ `for` syntax
- Example use of recursion
- Higher order functions

Higher Order Functions

Can define functions inline

```
cout << "Hello, "
     << ([](string x) -> string { return x; })("World!");
```

Higher Order Functions

Can define functions inline

```
cout << "Hello, "
     << ([](string x) -> string { return x; })("World!");
```

Capture list (explained next)

Higher Order Functions

Can define functions inline

```
cout << "Hello, "
     << ([](string x) -> string { return x; })("World!");
```

Argument list

Capture list (explained next)

Higher Order Functions

Can define functions inline

```
cout << "Hello, "
     << ([](string x) -> string { return x; })("World!");
```

Argument list

Return type

Capture list (explained next)

Higher Order Functions

Can define functions inline

```
cout << "Hello, "
     << ([](string x) -> string { return x; })("World!");
```

Capture list (explained next)

Argument list

Return type

Body

```
cout << "Hello, "
     << ([](string x) -> string { return x; })("World!");
```

Transform into…

```
// Lift inline defn here
string foo(string x) {
  return x;
}

// …
cout << "Hello, " << foo("World!");
```

Can you spot the problem?

```
string world = "World!";
([](string x) -> string { return x.append(world); })("Hello, ");
```

Think about what would happen if I were to do the foo trick again…

```
string world = "World!";
([](string x) -> string { return x.append(world); })("Hello, ");
```

Transform into…

```
string foo(string x) {
  return x.append(world);
}
```

What is the value of world here!?

Instead we want…

```
string foo(string x, string world) {
  return x.append(world);
}
```

Instead we want…

```
string foo(string x, string world) {
  return x.append(world);
}
```

Which we can write as…

```
([=] (string x) -> string { return x.append(world); })("Hello, ");
```

Instead we want...

```
string foo(string x, string world) {
  return x.append(world);
}
```

Which we can write as...

```
([=] (string x) -> string { return x.append(world); })("Hello, ");
```

Capture all the variables inside body that aren't x

We say the argument list *binds* x, things that aren't bound are called *free*

Capture list specifies which *free variables* to capture

```
vector<string> transformedArgs;

// This is the magic...
transform(arguments.begin()
          ,arguments.end()
          ,back_inserter(transformedArgs)
          ,[] (string x) -> string {return capitalize(x);});
```