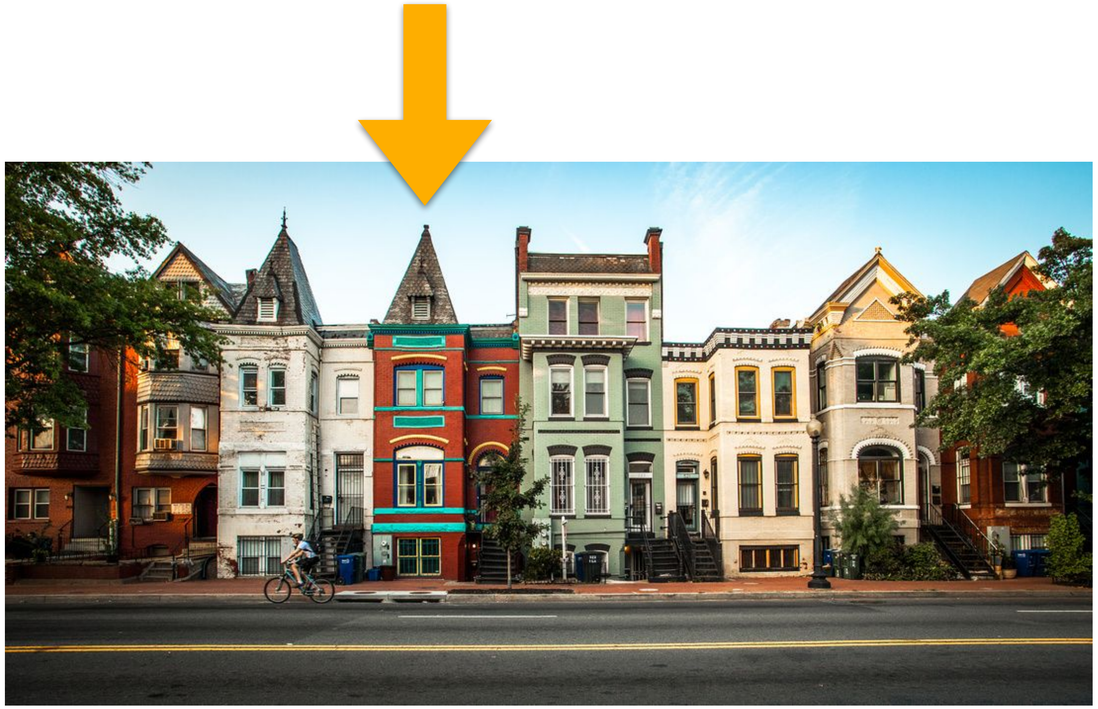
THE UNIVERSITY OF VERMONT
COLLEGE OF ENGINEERING &
MATHEMATICAL SCIENCES

Pointers and Addresses

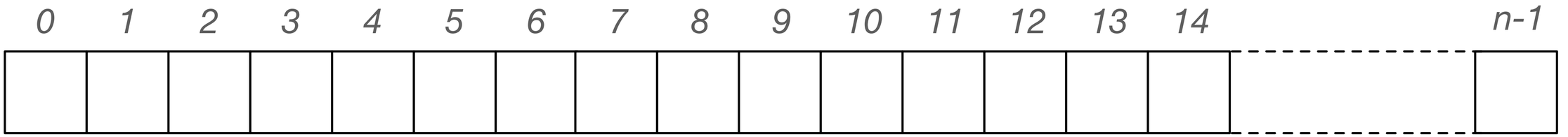


CS 124 / Department of Computer Science

Photo credit: © Josh Bassett (https://joshbassettphoto.com)

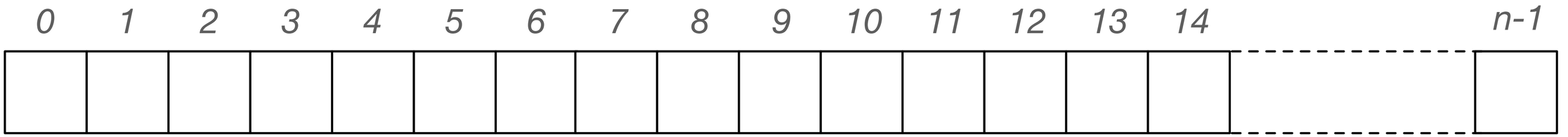
Memory addressing

Memory is a collection of locations, each of which can store one byte of information. Every memory location has an address. Addresses are sequential.



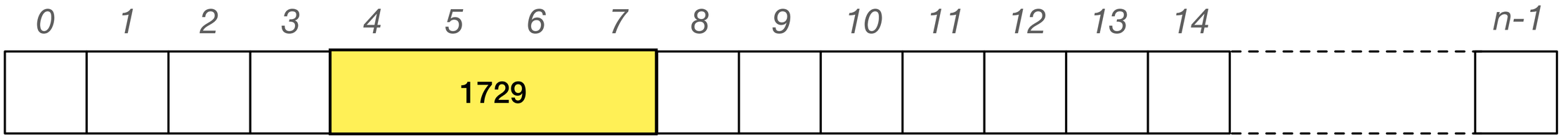
Memory addressing

When we store some variable to memory, an appropriately-sized portion of memory is reserved and the value is placed there.



Memory addressing

When we store some variable to memory, an appropriately-sized portion of memory is reserved and the value is placed there.



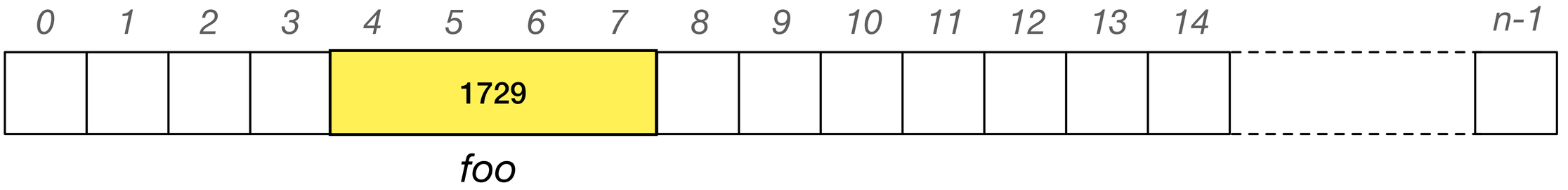
For example, an int takes four bytes of memory, since it can hold values from –2,147,483,648 to 2,147,483,647. Notice that 2,147,483,648 = 2^{32} / 2. So an int holds values in a range of size 2^{32}. Each byte is eight bits. So we need 32 bits, or four bytes to hold an integer. This amount of memory is reserved for each int regardless of its value. So all ints have the same size in memory.

Sizes of various data types in C++

int	4 bytes
short	2 bytes
long	8 bytes
long long int	8 bytes
float	4 bytes
double	8 bytes
char	1 byte
boolean	1 byte

Memory addressing

```
int foo = 1729;
```



Pointers: Motivation

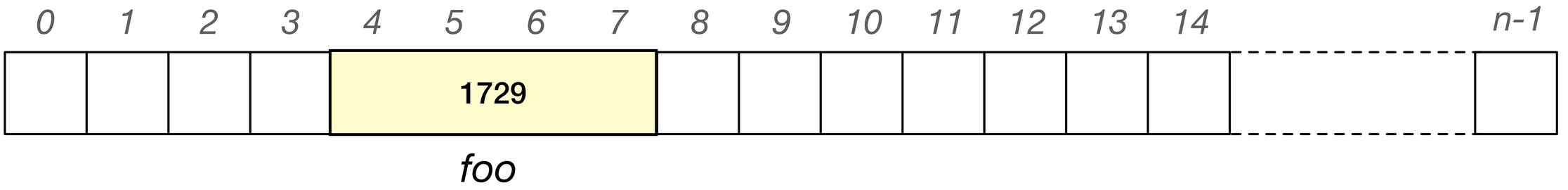
In general, we try not to use pointers, but there are times when they are handy or necessary.

In a little while, we'll learn about how to handle large objects, dynamic allocation, and objects that we wish to persist outside the scope in which they're declared. In these cases, use of pointers is necessary.

So here we're introducing pointers so they will be familiar to you when we need them — specifically when creating our node and stack classes.

Creating a pointer

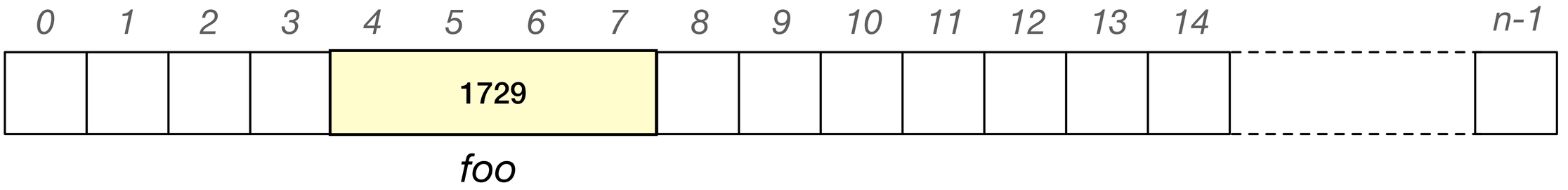
```
int foo = 1729;
```




```
int* fooPtr = &foo;
```

Creating a pointer

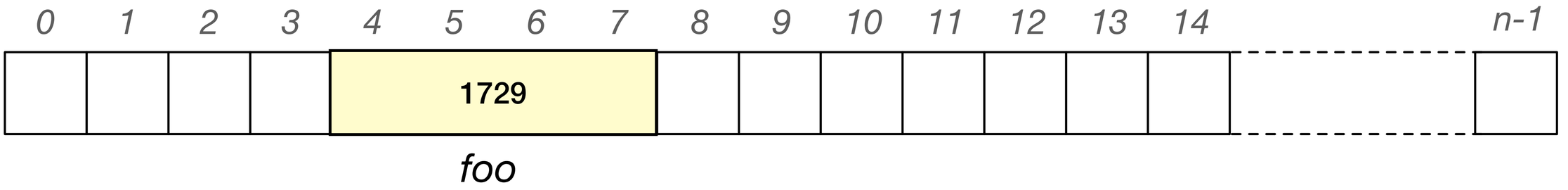
```
int foo = 1729;
```




```
int* fooPtr = &foo;
```

Creating a pointer

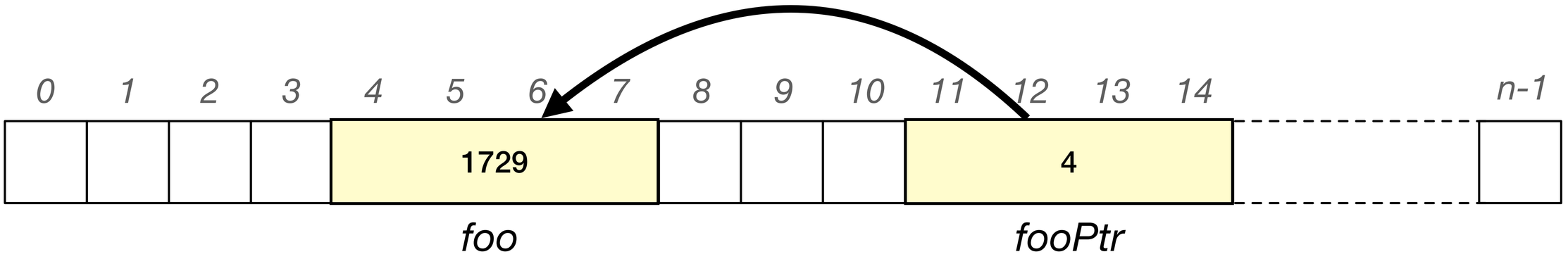
```
int foo = 1729;
```




```
int* fooPtr = &foo;
```

Creating a pointer

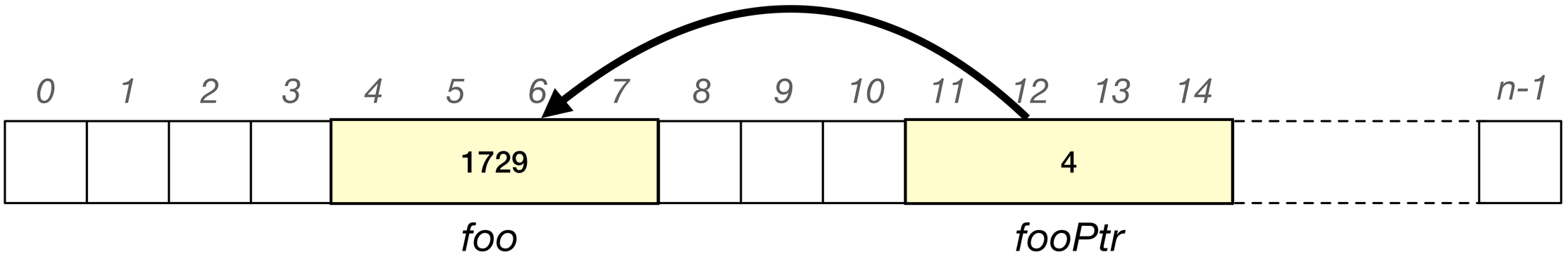
```
int foo = 1729;
```




```
int* fooPtr = &foo;
```

Creating a pointer

```
int foo = 1729;
```

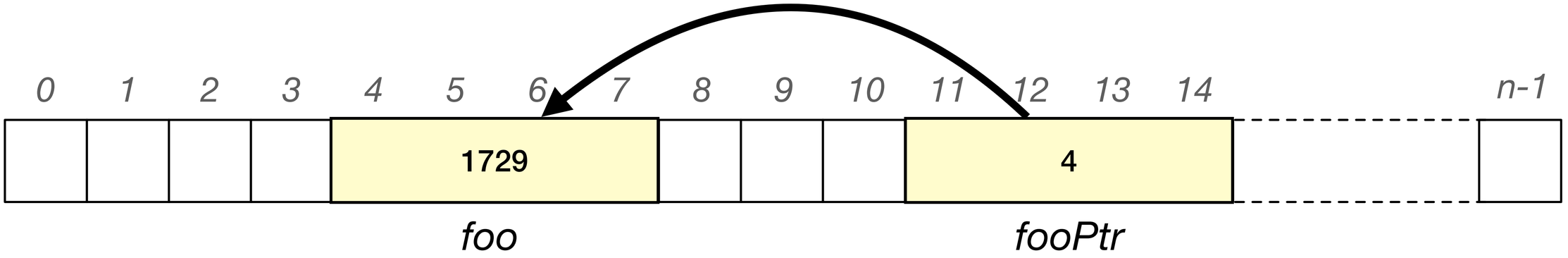



```
int* fooPtr = &foo;
```

Here we have the variable `foo`, which is an int, stored at address 4. The variable `fooPtr` stores the address of `foo` — it is a pointer to `foo`.

Dereferencing a pointer

```
int foo = 1729; int* fooPtr = &foo;
```

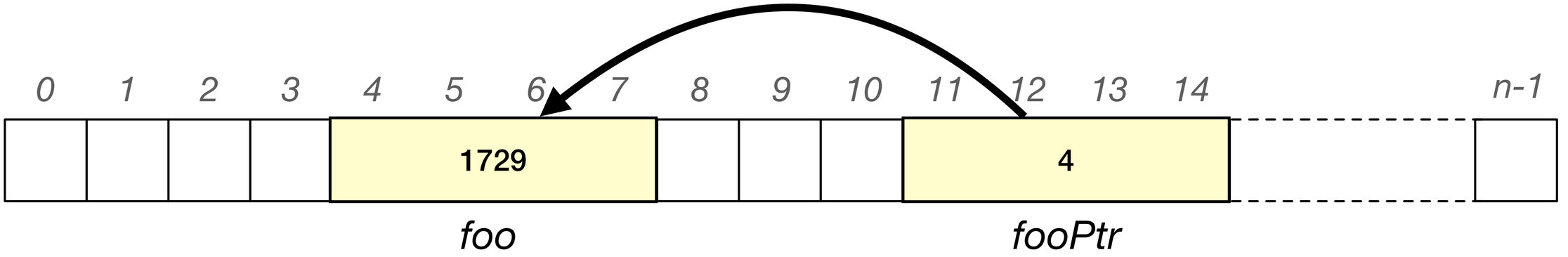



```
std::cout << *fooPtr << std::endl;
```

Prints "1729" — the value, or *the contents of*, the memory address pointed to by `fooPtr`.

Dereferencing a pointer

```
int foo = 1729; int* fooPtr = &foo;
```




```
std::cout << *fooPtr << std::endl;
```

Now you can see why we have to tell C++ the type that's being pointed to. How many bytes should the * operator retrieve when dereferencing? Unless C++ knows the data type, it can't know how many bytes to retrieve when dereferencing! Different data types have different lengths!

Sizes of various data types in C++

int	4 bytes
short	2 bytes
long	8 bytes
long long int	8 bytes
float	4 bytes
double	8 bytes
char	1 byte
boolean	1 byte

Pointer and address operators

You may read * as "content of" and & as "address of", hence:

```
int foo = 1729;
int* fooPtr = &foo;
std::cout << "The content of the address " << fooPtr
          << " is " << *fooPtr << std::endl;
std::cout << "The value " << foo
          << " is stored at " << &foo << std::endl;
```

Pointer and address operators

You may read * as "content of" and & as "address of", hence:

```
int foo = 1729;
int* fooPtr = &foo;
std::cout << "The content of the address " << fooPtr
          << " is " << *fooPtr << std::endl;
std::cout << "The value " << foo
          << " is stored at " << &foo << std::endl;
```

Prints...

```
The content of the address 0x7ffee56567c8 is 1729
The value 1729 is stored at 0x7ffee56567c8
```

What is 0x7ffee56567c8?

Numbers prefixed by `0x` in C++ are in hexadecimal notation — and this is the way pointers — memory addresses — are displayed. But you may think of this as an int. A large int to be sure, and an int that refers to a memory address, but an int nonetheless.

In C++ addresses are all the same size. This is determined by the size of the address space, typically either 32 or 64 bits.

Note that if you compile and run the code on the previous slide, you'll get a different hexadecimal address for `foo`.

Summary

- Every variable or object in C++ has some location in memory.
- We can create a pointer to any such variable or object by using the appropriate syntax: `<datatype of someVar>* someName = &someVar;` For example, we create a pointer to an integer `foo` with `int* fooPtr = &foo;`
- `*` is the "content of" unary operator
- `&` is the "address of" unary operator
- "Dereferencing" is a fancy word for "getting the contents of some address"