CS 112

Introduction to Data Structures

WEEK 0

Intro to C++

Eric Araújo

Calvin University · Fall 2026

This Week

1. Structure of a C++ program
2. Compilation pipeline
3. Types, variables, I/O
4. Functions
5. Pointers **KEY**
6. Arrays
7. C++ vs. Python

Your First C++ Program

```cpp
int main() {


    return 0;
}
```

> Every C++ program needs a `main()`: this is where execution begins.

Your First C++ Program

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

> Notice how `main()` morphed into its final form; we added the include, namespace, and output line.

Anatomy of a C++ Program

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

1 `#include <iostream>`

Pulls in the I/O library, C++'s version of `import`.

2 `using namespace std;`

Lets you write `cout` instead of `std::cout`.

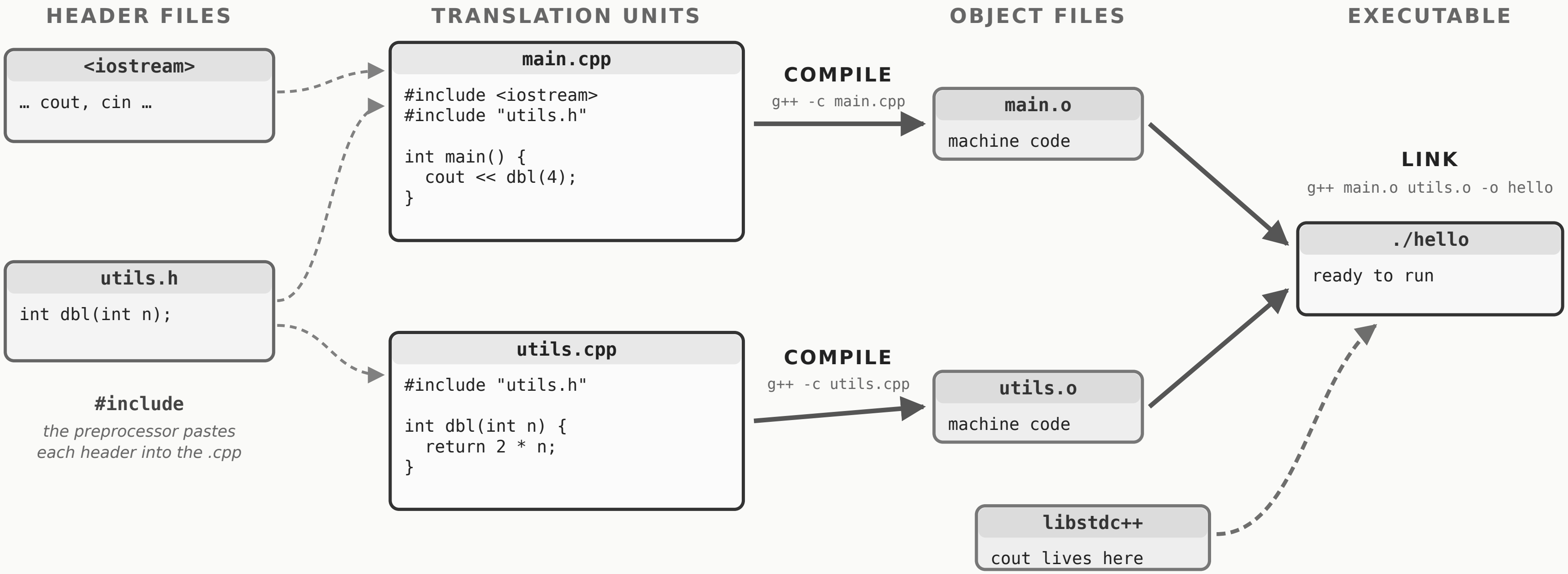
5 `cout << … << endl;`

Prints to the terminal. `<<` is the “injection” operator. `endl` ends the line *and* flushes the buffer, so the text appears immediately.

6 `return 0;`

Tells the OS the program finished cleanly.

Compilation Pipeline



COMPILE-TIME ERROR

`error: 'cout' was not declared in this scope`

One file didn't make sense on its own: a typo, a missing `#include`, a type mismatch.

LINK-TIME ERROR

`` undefined reference to `helper()' ``

Every file compiled fine, but a function was *declared* and never *defined*.

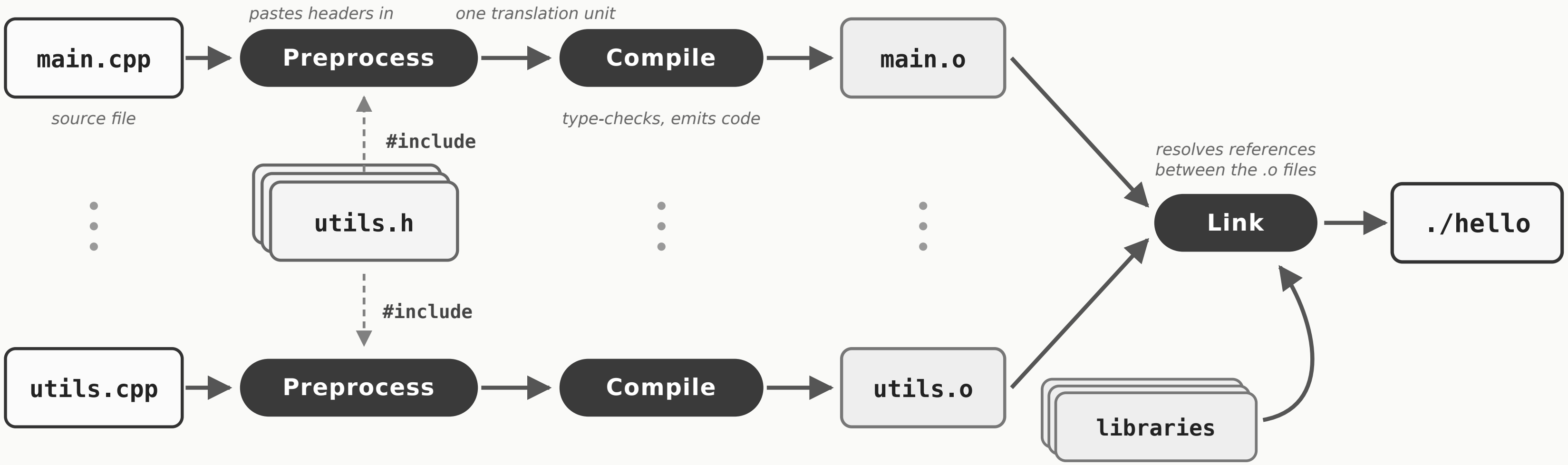
Headers and Translation Units

`header .h`

Declarations: what exists and how to call it. Shared by many `.cpp` files, and never compiled on its own.

`translation unit .cpp`

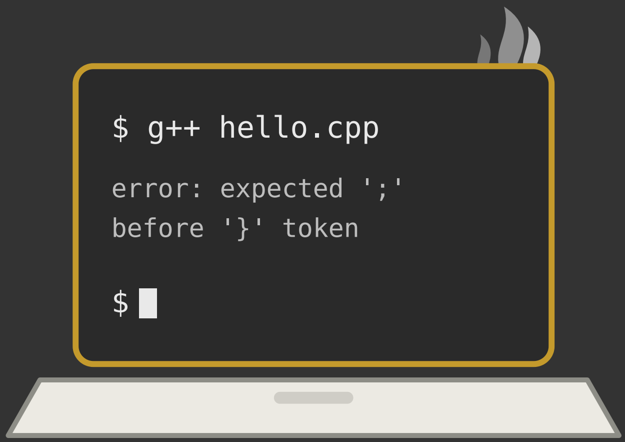
One `.cpp` *after* its headers have been pasted in. This is what the compiler actually reads, and it becomes exactly one `.o`.



Unlike Python, you *must* compile before running.

Upside: a whole class of bugs is caught at compile time, not in production.

Demo



Live coding. What could possibly go wrong?

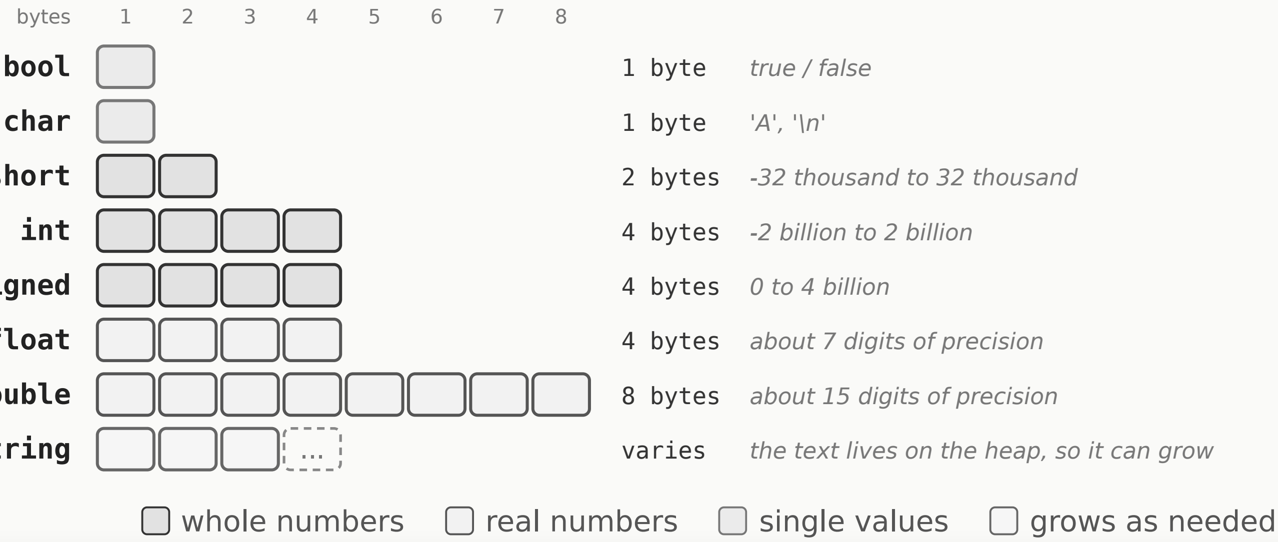
(the semicolon is always missing on line 12)

TALK TO YOUR NEIGHBOR · TTYN

What does a compiler do?

- **A.** Create an executable program from source code.
- **B.** Check the source code for errors.
- **C.** Pull in required libraries.
- **D.** A and B only.
- **E.** B and C only.
- **F.** A, B, and C.
- **G.** None of the above.

Data Types & Sizes



bool	1 byte	*true / false*
char	1 byte	*'A', '\n'*
short	2 bytes	*-32 thousand to 32 thousand*
int	4 bytes	*-2 billion to 2 billion*
unsigned	4 bytes	*0 to 4 billion*
float	4 bytes	*about 7 digits of precision*
double	8 bytes	*about 15 digits of precision*
string	varies	*the text lives on the heap, so it can grow*

```
cout << sizeof(int);        // 4: check any type yourself
cout << sizeof(double);     // 8
cout << sizeof(grade);      // 1: works on variables too
```

Types are **fixed at compile time**.

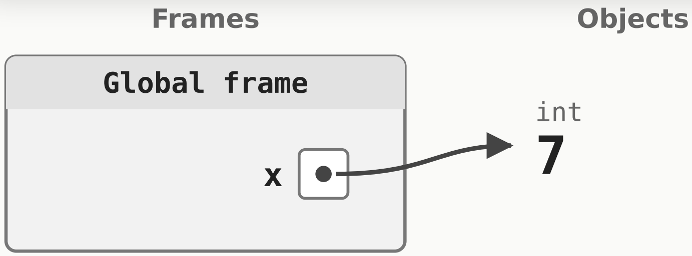
Sizes can vary by machine, `sizeof` tells you the truth on yours.

A `string` keeps its characters on the **heap**, memory the program asks for while it runs. More on that in week 4.

Python vs. C++: Variables

Python

```python
x = 7
```



the name is a label; the object knows its own type

C++

```cpp
int x = 7;
```

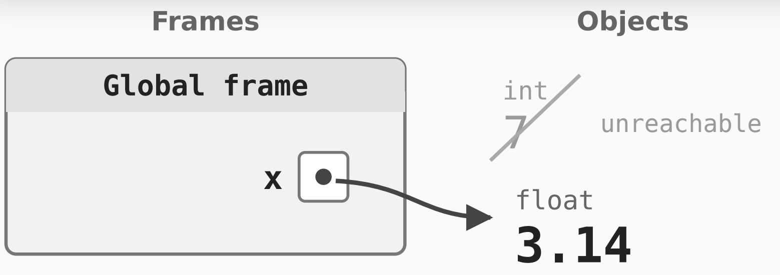
the box is on the stack, with a fixed type and address

> Both look like "a variable called x holding 7".
> In C++ that box lives on the **stack**, inside the *stack frame* belonging to this call of `main()`. Watch what happens when we assign `x = 3.14`.

The Same Variable, Reassigned

Python

```
x = 3.14
```



same name, new object, new type

C++

```
x = 3.14;
```

same box on the stack: same address, same type

> Python moves the label: `x` points at a *new* float object, and the `int 7` is abandoned.
> C++ keeps the box: same address, still an `int`, so 3.14 is truncated to **3**.

Declare, Then Initialize

```
...
int    count;              // declared: NOT initialized
int    total  = 0;         // declared and initialized
double pi     = 3.14159;
char   grade  = 'A';
bool   ready  = true;
string name   = "Ada";

cout << "count: " << count << endl;   // ⚠ undefined: could print anything
cout << "total: " << total << endl;   // always 0
...
```

```
$ ./init_demo
count: -8593204                              whatever those bytes happened to hold
total: 0                                     exactly what you put there
$ ./init_demo                                same program, run again
count: 32764                                 different garbage this time
total: 0
```

> In Python a name doesn't exist until you assign to it.
> In C++ the box exists the moment you *declare* it, holding whatever the last program left in that memory.

TALK TO YOUR NEIGHBOR · TTYN

Will this C++ code compile?

```
int x = 11;
x = "12";
```

A. Yes

B. No

C. Yes, but with warnings.

Using `const`

```
const int    MAX_SIZE = 100;
const double PI       = 3.14159265;

MAX_SIZE = 200;  // □ compiler error: read-only!
```

- Put `const` before the type to make a variable read-only
- The compiler **rejects** any code that tries to change it
- Convention: use `ALL_CAPS` names for constants

> Use `const` liberally; it documents intent and catches bugs at compile time.

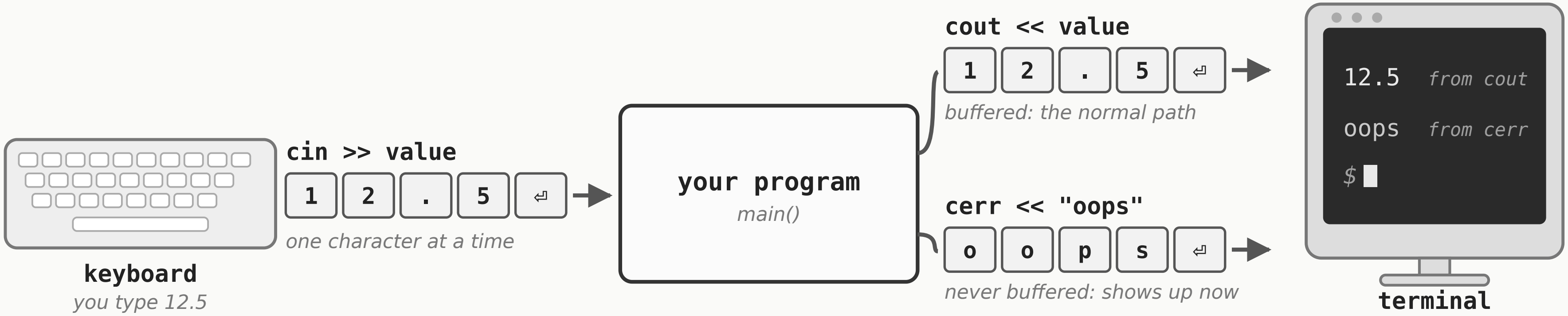
Input / Output

```
#include <iostream>
using namespace std;

int main() {
    double value;
    cout << "Enter a number: ";   // output → terminal
    cin  >> value;                // input  ← terminal
    cerr << "got " << value << endl; // error stream

    cout << "You entered: " << value << endl;
    return 0;
}
```

`cout` + `<<`: **injection**
output to the terminal

`cin` + `>>`: **extraction**
input from the terminal

Streams



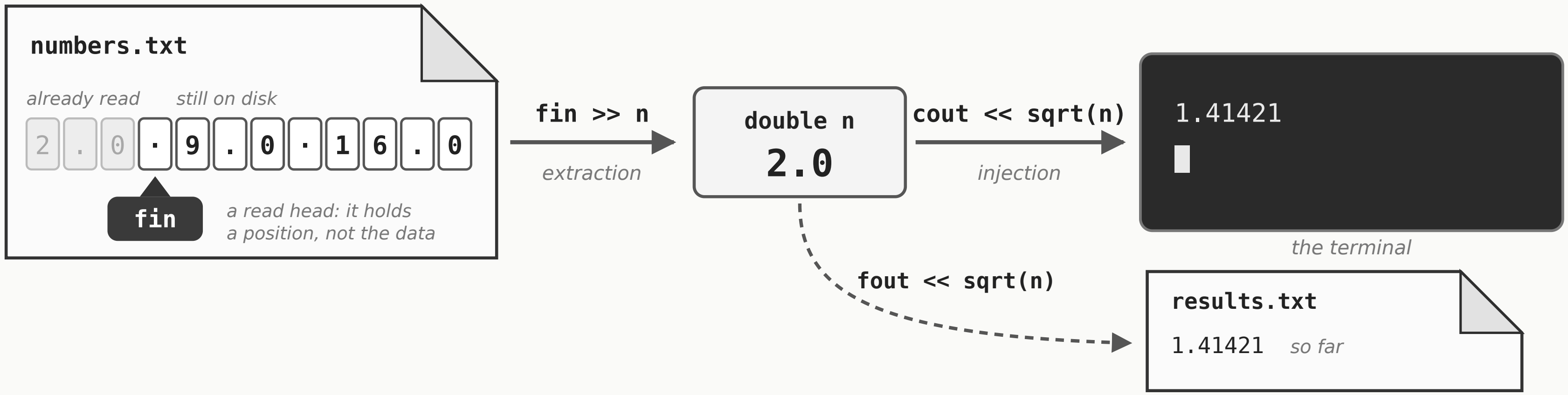
A stream is a **one-way sequence of characters**, not a function call.
`cin` flows in, `cout` and `cerr` flow out.
Swap the keyboard for a file and the picture does not change: that is `ifstream`, and it is the next slide.

File I/O

```
#include <iostream>
#include <fstream>    // ifstream reads a file, ofstream writes one
#include <cmath>      // sqrt
#include <cassert>
using namespace std;

int main() {
    ifstream fin("numbers.txt");  // open the file, attach a stream to it
    assert(fin.is_open());        // a missing file is not an error to ignore

    double n;                     // ONE variable, reused for every number
    while (fin >> n) {            // read the next number; false when none left
        cout << sqrt(n) << endl;  // fin has already moved past that number
    }

    fin.close();                  // hand the file back to the OS
    return 0;
}
```

> `ifstream` reads from a file exactly like `cin` reads from the terminal. The loop ends on its own, because `fin >> n` is false once the file runs out.

What `fin >> n` Actually Does



The data never leaves the file: `fin` is a **read head** parked in it. Every `>>` converts the next characters into a `double` and drags that head forward. `<<` runs the other way, and does not care where it is pointed: `cout` for the screen, an `ofstream` for a file.

Decimal, Binary, Hexadecimal

```
int x = 33;         // decimal  (default)
int y = 0x21;       // hex:     2×16 + 1 = 33
int z = 0b100001;   // binary: 1×32 + 1 = 33

cout << x << " " << y << " " << z << endl;
// Output:  33 33 33
```

- `0x` prefix → hexadecimal (base 16)
- `0b` prefix → binary (base 2)
- Memory addresses are typically displayed in hex

> Three ways to write the same number *same number*. The compiler stores them identically in memory.

Meet Rocky

Not this Rocky.

Prof. Rocky Chang, our department

THIS Rocky!

Rocky, an Eridian, from Project Hail Mary

Why Ten?

Three limbs down, two up: two hands of three fingers, so he counts in sixes.

how many things	0	1	2	3	4	5	6	7	8	9	10	11	12
we count in tens	0	1	2	3	4	5	6	7	8	9	**10**	11	12
Rocky counts in sixes	0	1	2	3	4	5	**10**	11	12	13	14	15	20
computers count in twos	0	1	**10**	11	100	101	110	111	1000	1001	1010	1011	1100

Ten is an accident of anatomy: we have ten fingers.
Rocky stands on three of his five limbs, leaving two hands of three fingers, so Eridians count in **sixes**: their "10" is our 6.
A wire is either on or off, so computers count in **twos**, and hex packs four of those bits into one digit.

Why bother with binary?

Everything on the machine is bits. Hex is just a friendlier way to read them.

xkcd #99 · xkcd.com · CC BY-NC 2.5

Functions

Write the tests *before* the function; it forces you to think about what it should do:

```
assert(factorial(0) == 1);
assert(factorial(1) == 1);
assert(factorial(4) == 24);
assert(factorial(5) == 120);
```

> This test-first style is called **test-driven development**. Your assertions become the spec.

Functions

```
unsigned factorial(unsigned n) {
    unsigned result = 1;
    for (unsigned i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}

int main() {
    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
    assert(factorial(4) == 24);
    assert(factorial(5) == 120);
    cout << "All tests passed!" << endl;
    return 0;
}
```

Pointers

Memory is a row of numbered boxes. A variable is one box.

```c
int y = 77;
```

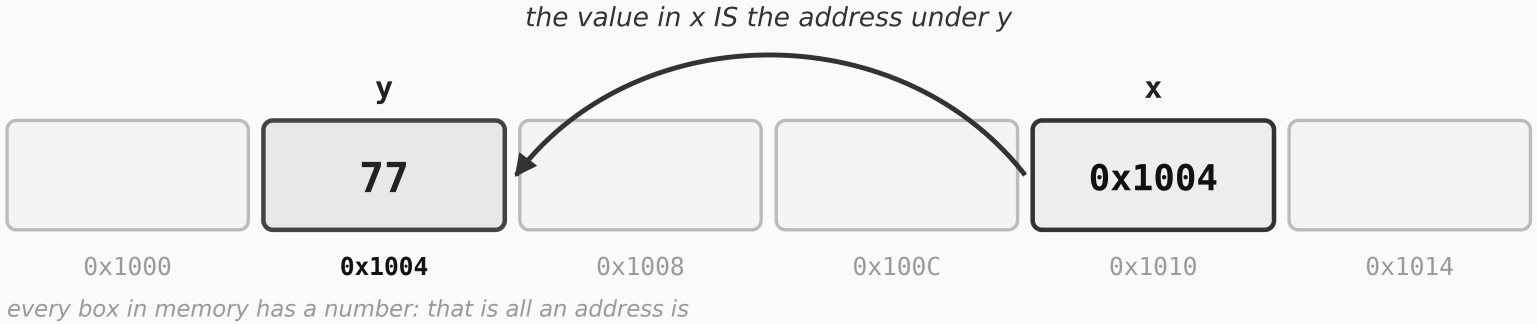
every box in memory has a number: that is all an address is

> `y` is the box at `0x1004`. The name is ours; the machine only knows the number.

Pointers

`&` ("address-of") hands you the number of a box.

```
int y  = 77;
int *x = &y;   // x holds the address of y
```

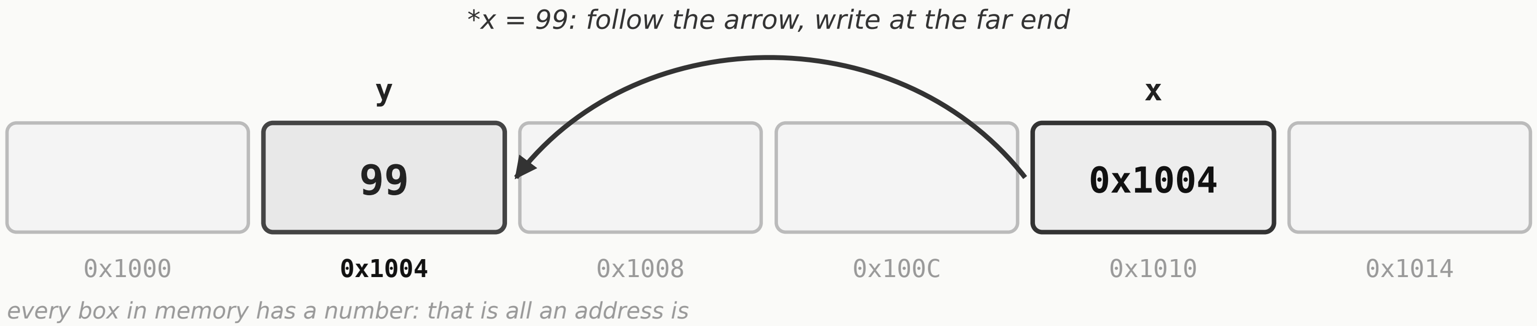


> `x` is an ordinary box too. What makes it a pointer is *what it holds*: an address.
> The arrow is us drawing the same number twice. The machine just has `0x1004`.

Pointers: Dereferencing

`*` means "go to that address".

```
int y  = 77;
int *x = &y;

cout << x;    // 0x1004: the address itself
cout << *x;   // 77:     follow the arrow
*x = 99;      // write at the far end of the arrow
cout << y;    // 99:     y changed, without naming y
```



> `x` and `*x` are different questions: *which box* versus *what is in it*.

TALK TO YOUR NEIGHBOR · TTYN

```
int y = 77;    // placed at address 0x1004
int *x = &y;  // placed at address 0x1010
cout << x;
```

What is printed?

A. 77

B. 0x1004

C. 0x1010

D. Error

TALK TO YOUR NEIGHBOR · TTYN

```
int y = 77;
int *x = &y;
cout << *y;   // note: *y, not *x
```

What is printed?

- **A.** 77
- **B.** 0x12345678
- **C.** 0x1010
- **D.** Compile error

TALK TO YOUR NEIGHBOR · TTYN

```
int y = 77;
int *x = &y;
cout << *x;
```

What is printed?

A. 77

B. 0x12345678

C. 0x1010

D. Error

TALK TO YOUR NEIGHBOR · TTYN

```
int y = 77;
int *x = &y;
*x = 78;
cout << *x;
```

What is printed?

- **A.** 77
- **B.** 78
- **C.** 0x1010
- **D.** Error

TALK TO YOUR NEIGHBOR · TTYN

```
int y = 77;
int *x = &y;
*x = 78;
cout << y;    // note: y, not *x
```

What is printed?

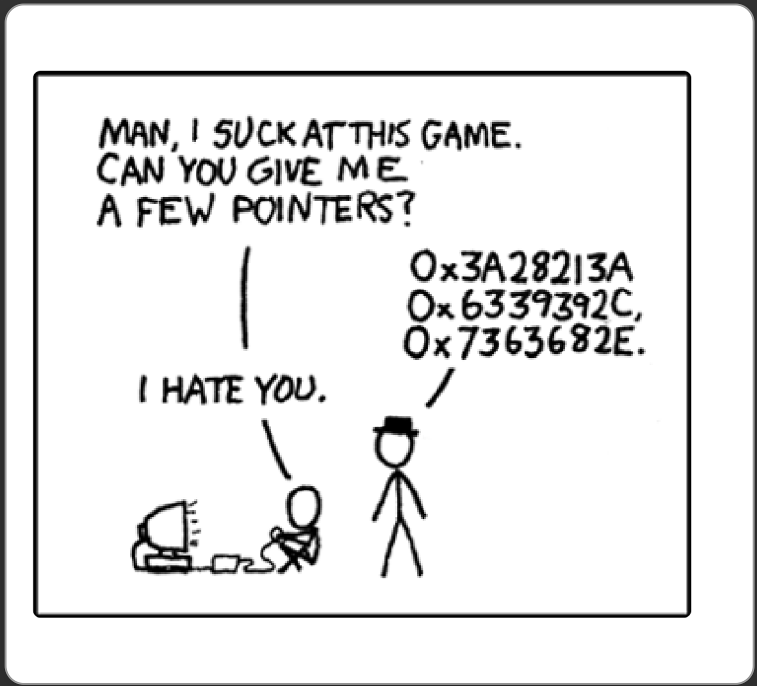
A. 77

B. 78

C. 0x1010

D. Error

Ask a programmer for a few pointers



He did give him pointers. Three of them, and every one is a real address.

xkcd #138 · xkcd.com · CC BY-NC 2.5

Arrays

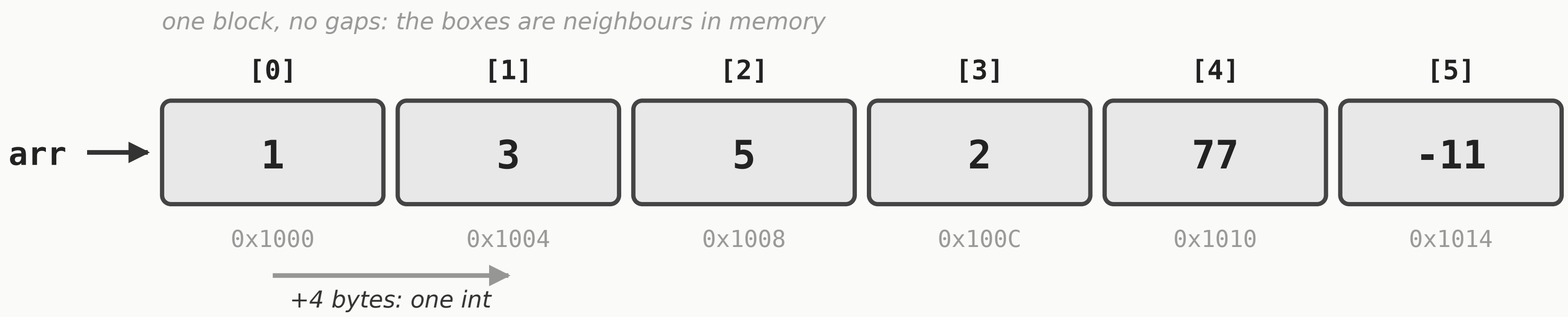
Like Python lists, but **fixed size** and **no bounds checking**.

```
char  letters[26];                // 26 chars, uninitialized
float scores[10] = { 0 };         // all 0.0
int   vals[] = { 10, 9, 4, 2, 0 };  // size inferred: 5
```

- Size must be known at compile time (for stack arrays)
- Indexes: `0` to `n−1`
- Accessing out of bounds → **undefined behavior**, no crash guaranteed

Arrays: Memory Layout

```
int arr[6] = { 1, 3, 5, 2, 77, -11 };
```



`arr` is a pointer to the first box, so `arr[i]` means "start at `arr`, step `i` ints along".
That is why indexes start at **0**: the first element is zero steps from the start.

Pointers & Arrays

```
int arr[] = { 3, -11, 4, 12 };
int *ptr  = &arr[2];                 // ptr points at arr[2]

cout << *ptr       << endl;         // this box
cout << *(ptr + 1) << endl;         // one int further along
cout << *(ptr - 1) << endl;         // one int back
```

```
$ ./ptr_demo
4                                   arr[2], where ptr points
12                                  arr[3], the next box
-11                                 arr[1], the box before
```

> `ptr + 1` does not add 1 to the address: it adds **one int**, so 4 bytes.
> That is exactly what `arr[i]` has been doing all along.

TALK TO YOUR NEIGHBOR · TTYN

```
int arr[] = { 3, -11, 4, 12 };
int *ptr  = &arr[2];
cout << *ptr;
```

What is printed?

A. 3

B. 2

C. 4

D. -11

C++ vs. Python: Core Differences

Feature	Python	C++
Variables	Reference to a typed object	Named memory location with explicit type
Type checking	Dynamic (runtime)	Static (compile time)
Function return type	Implied from `return`	Declared explicitly
1D data structure	`list`	`array` or `vector`
Memory management	Automatic (garbage collected)	Mostly explicit
Entry point	First line executed	`main()`

C++ vs. Python: More Differences

Feature	Python	C++
Interface / impl split	Not explicit	`.h` header + `.cpp` source
Parameter passing	Pass-by-value (objects by ref)	By-value, by-reference, by-const-ref
Access control	Mostly none	`public` / `private`
Constructors	Called explicitly	Explicit *or implicit* (scope entry)
Destructors	None (GC handles it)	Defined; called implicitly at scope exit
Generics	None	Templates, powerful but complex
Strings	Built-in	`#include <string>`

Week 0 Recap

- C++ programs need a `main()` and must be **compiled** before running
- Types are **explicit and checked at compile time**
- Pointers hold **memory addresses**: the key concept in C++
- Arrays are contiguous memory blocks: fast, but no safety net
- C++ gives you more control than Python, and more responsibility

Next: Getting Started (pre-lab) · then a01: RPG Character Creator →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**