C & C++ Crash Course

https://github.com/evertrol/cccc2017

Evert Rol - Monash University - February 2017
evert.rol@monash.edu

Why use C or C++

- Familiarity
- Existing code / framework
- Speed
- (Near) complete control: systems programming language
- Integration with other languages (e.g., the standard Python usually used is coded in C)
- GPU programming: OpenCL / CUDA are very C-like

Why not?

- **Safety**
- Cumbersome to code (e.g., no vector-style `x = log(y)`)

C and C++ compared

Essentially 3 (2½) languages:

- C very steady over time
  Few changes last decades
  (but do use the C99 or C11 standard)
- Old-style C++ (C++98)
  Much used still, but essentially C with classes
- Modern style C++ (C++11 / C++14)
  Quite different & safer.
  Use when you can (may depend on existing code / compiler)

Don’t view C++ and C as the same language!

Apologies for the caveats and warnings

- There will be quite a few mentions of problems you can encounter
- Not to scare you; just to be aware of it (especially when having to modify existing, older, code)
- Modern style C++ (C++11 / C++14)
  Quite different & safer.
  Use when you can (may depend on existing code / compiler)

Software needed

Ubuntu / Debian:

```
apt-get / apt   install build-essential
```

Fedora / Redhat:

```
yum / dnf   install build-utils
```

macOS:

```
xcode-select --install
```

(for extras on macOS, I use Homebrew as package manager)

Test with:

```
gcc --version
g++ --version
make --version
```

Anatomy of a C program

```
#include <stdio.h>
#include <stdlib.h>
#define N 1000


int main()
{
    int sum = 0;
    for (int i = 1; i <= N; ++i) {
        sum += i;
    }
    printf(“%d\n”, sum);

    return EXIT_SUCCESS;
}
```

#: Preprocessor directives
These are processed before compilation to machine code

Include directive: include this file into the program

<…> a “system” include file that the compiler automatically finds

stdio.h: includes the definition of printf function
stdlib.h: includes the definition of EXIT_SUCCESS constant

#define a macro: this gets replaced during preprocessing

N gets replaced by 1000. Use capitals for macros (constants)

Anatomy of a C program

```
#include <stdio.h>
#include <stdlib.h>
#define N 1000

int main()
{
    int sum = 0;
    for (int i = 1; i <= N; ++i) {
        sum += i;
    }
    printf("%d\n", sum);

    return EXIT_SUCCESS;
}
```

main() function: entry point of any C/C++ program
It returns an integer

Declare an integer, and set its value

Semi-colon at the end of a line.
A newline does *not* end a statement

Curly braces to indicate a block

Core: do the actual calculation

`printf` is a library function (lib-c)
Use for output to console

`main` should return an integer.
EXIT_SUCCESS is clearer than 0.

Compile and run a C program

C compiler: `gcc` / `clang`
(C++ compiler: `g++` / `clang++`)

`-Wall -Wextra -Wpedantic`
(`clang` has `-Weverything`)

Use compiler warnings flags
Read the compiler warnings and errors

`-std=c11 /  -std=c99`
`C++: -std=c++14 /  -std=c++11`
`     (-std=c++1z / -std=c++17)`

Use a (modern) standard

Then your compilation becomes:

```
gcc -Wall -Wextra -Wpedantic -std=c11 program1.c -o program1
```

C programs simply have `.c` as an extension
C header files have `.h` as an extension

Compile and run a C program

If you use libraries, use the `-l` flag with the library name.
For example, the math library is simply called `m`:

```
gcc -Wall -Wextra -Wpedantic -std=c11 program.c -o program -lm
```

NB: the C library (called `c`) is implicit: no need for `-lc`

Behind the scenes, three steps happen:

- The C preprocessor is run: files are included, macros are expanded
- The file is compiled to machine code (an object file)
- The object file is linked with libraries into an executable

Usually, you don't care about this, but it is good to know that if things go wrong, at what stage an error occurred.

Compile and run a C program

```
gcc -Wall -Wextra -Wpedantic -std=c11 program.c -o program -lm
```

Too much to type?

Create a file called `Makefile`:

```
CC=gcc
CFLAGS=-Wall -Wextra -Wpedantic -std=c11
LDFLAGS=

main: program1.c
⇥$(CC) $(CFLAGS) program1.c -o program1 $(LDFLAGS)

clean:
⇥rm -f *.o program1
```

Now simply run `make` on the command line

You can change the variables inside the Makefile, or temporarily on the command line. For example:

```
make CC=clang LDFLAGS=-lm
```

Exercise

Implement (*type*, don't copy & paste), compile and run the summation example program

Fibonacci sequence

The Fibonacci sequence is given by

$$F_n = F_{n-1} + F_{n-2}$$

with

$$F_0 = 0, F_1 = 1$$

Using the summation program as example, implement the Fibonacci sequence calculation.

- Print each term (both index and value) inside the loop
- First, print only the first 5 or 10 numbers to verify the calculation
- Next, print the first 50 Fibonacci numbers

Types

C & C++ need to have variable types declared

Available integer types:

`char`: single byte value (0-225). Used for single (ASCII) characters, or small integer values

`short (int)`: *at least* $–2^{15}$-1 to $+2^{15}$-1   (32767)

`int`: *at least* $–2^{15}$-1 to $+2^{15}$-1   (on 64-bit systems, usually 2^{31}-1)

`long (int)`: *at least* $–2^{31}$-1, $+2^{31}$-1   (2147483647)

`long long (int)`: *at least* $–2^{63}$-1, $+2^{63}$-1   (9,223,372,036,854,775,807)

All can also be `unsigned` (default is `signed`). Normally, use signed.

Types

Available floating point types:

`float`: $\approx 10^{-38}$ to $\approx 10^{38}$, with ≈ 7 digits precision.

`double`: $\approx 10^{-308}$ to $\approx 10^{308}$, with ≈ 15 digits precision.

`long double`: even more precision, but less standardized.
Use with caution.

The float and double values are based on the IEEE 754 standard, which is usually used.
These types are always signed.

Types

Each type has a conversion specifier in printf:

```
int:            printf(“%d”, value);
long:           printf(“%ld”, value);
long long:      printf(“%lld”, value);
double/float:   printf(“%f”, value);
double/float:   printf(“%e”, value);  // exponential notation
```

Types

No boolean type in C
Only available when including `<stdbool.h>`
A bool is essentially an integer
  (`0 == false, <everything else> == true`)

No string type in C
A string in C is essentially an array of characters.
And a source of errors…

Better in C++, thanks to its standard library

Exercise: compiler optimization

Back to the summation program

- Wrap the summation in a loop itself, and reset `sum` inside the outer loop, so it runs longer
- Find the number of iterations (inner times outer) where the program runs just a few seconds, without optimizations.
  You can use the bash / Linux function `time`: `time <program>`
  (Using `time` is not precise, but a good approximation for a few seconds)
- Now compile with `-O3` (full optimization), and do the same
- Now use a non-trivial summation, e.g.: `sum += log(i);`
  Use `-O3`, **`#include <math.h>`** and link with libm (`-lm` flag)
- Now remove the final `printf` function, and do the same

Compiler optimization

- Compilers are smart!
- Be wary of premature and micro optimizations
- -O0 behaviour (default) ≠ -O3 behaviour
  Quite a few things happen to be zero at -O0, but they don’t have to be, and usually aren’t at -O3. This becomes more important when dealing with pointers, arrays and strings

if-else / for / while

```
if (x < 0) {
    x = -x;
}
```

```
if (x < 0) {
    x = -x;
} else {
    x += 3;
}
```

```
if (x < 0) {
    x = -x;
} else if (x == 0) {
    x -= 3;
} else {
    x /= 2;
}
```

Requires parentheses around the boolean expression
Don’t do anything fancy inside them: `if (x = 5 < 10) { …`

if-else / for / while

Set once just before the loop starts
i is only visible in the loop (scope)

Tested at the *start* of each iteration

```
for (int i = 0; i < 10; ++i) {
    <something>
}
```

Performed at the *end* of each iteration

```
for (int i = 9; i >= 0; --i) {
    <something>
}
```

```
for (double x = -3.14; x < 20; x += 0.1) {
    <something>
}
```

```
for (int x = 2; x <= 1024; x *= 2) {
    <something>
}
```

if-else / for / while

```
int i = 0;
while (i < 10) {
    <something>
    ++i;
}
```

```
#include <stdbool.h>
while (true) {
    <something>
    if (<whatever>) {
        break;
    }
    <other stuff>
}
```

`break`: break out of the current loop (only one level)

`continue`: skip to start of the next loop iteration (it *does* perform the increment part in a for-loop!)

i++ / ++i / i-- / --i

These are not exactly shortcuts for `i += 1` or `i -= 1`

Prefix: compute first, then evaluate

```
    int i = 500;
    if (++i > 500) { … }
```

Postfix: evaluate first, then compute

```
    int i = 500;
    if (i++ > 500) { <never gets here> }
```

Use them in a single statement / single line ***only***

The examples below are fun, but more than confusing:

```
    for (int i = 10; i --> 0; ) {
    }

    i = 1;
    i = i++ + ++i;      // undefined behaviour
    i == ?
```

Braces / indentation / comments

Always use braces, even in loops and if-statements with one statement.

```
for (int i = 0; i < 10; ++i) {
     x += i;
}

if (x > 10) {
    x -= 10;
}
```

Allowed, but leads to errors in the long run

(infamous Apple SSL bug)

Braces / indentation / comments

Always use braces, even in loops and if-statements with one statement.

```
for (int i = 0; i < 10; ++i)
if (i > 3)
if (i < 8)
x += i;
else if (i > 1)
x += 2*i;
else
x += 3*i;
```

And may be utterly unreadable

```
for (int i = 0; i < 10; ++i)
  if (i > 3)
    if (i < 8)
      x += i;
    else if (i > 1)
      x += 2*i;
  else
    x += 3*i;
```

```
for (int i = 0; i < 10; ++i)
  if (i > 3)
    if (i < 8)
      x += i;
    else
      if (i > 1)
        x += 2*i;
      else
        x += 3*i;   // never gets here
```

Braces / **indentation** / comments

Indentation is just good programming practice

Linus Torvalds suggest using tabs (width of 8 spaces)
This avoids, for example, nesting loops and if-statements too deeply
(especially if you also use 80-column widths)

In general, use the current project style

Braces / indentation / **comments**

Use one-line comments for remarks on a piece of code:

```
// Take care things don't get unphysical
if (x < 0) {
     x = 0;
}
```

It's perfectly fine to have a few lines of `//` one-line comments

Don't comment the obvious:

```
i++;  // increment i by 1
```

Braces / indentation / **comments**

Use comment blocks for longer explanations, for example function descriptions

```
/*  The below function calculates the nth term
    of the Fibonacci sequence
 */
```

You can use comment blocks to temporarily comment-out pieces of code by surrounding it with `/*` and `*/`

```
/*
if (x < 10) {
  x += 10;
}
*/
```

Be aware that block comments don’t nest!

Functions

Divide and ~~conquer~~* keep it readable / reusable

```
/*
 * Description of the function
 * - what it does and returns
 * - description of its arguments
 * - algorithm used
 * - side effects
 * - valid input range
 * - possible errors
 */
<return type> <function name> (type name, type name2)
{
        <calculations>

        return <value>;  // not all functions return a value, and a few
                         // don’t return at all
}
```

* This is actually a type of algorithm

Functions

Divide and ~~conquer~~ keep it readable / reusable

```
/*
 * Fibonacci(int nterm)
 * - Calculates and returns the nterm-th value of the Fibonacci sequence
 *
 * - The value is calculated by evaluating the Fibonacci sequence:
 *   Fn = Fn-1 + Fn-2
 *    with F0 = 0 and F1 = 1
 * - Argument nterm should be between 0 and …
 *    - the sequence is not defined for negative terms
 *    - at input values above … overflow occurs
 */
long long fibonacci(int nterm)
{

        return value;
}
```

Functions

Divide and ~~conquer~~ keep it readable / reusable

```
#include <stdlib.h>

int main()
{
    return EXIT_SUCCESS;  // 0 on most systems
}
```

is a special function: it's the entry point of any C/C++ program.

The return value is returned to the system. In bash, you can see this return value with

```
    echo $?
```

once the program has finished

(replace `EXIT_SUCCESS` with `EXIT_FAILURE` and find out what value it has on your system)

Exercise

- Put the Fibonacci calculation inside a function
  Call it from main, and print its return value there
- Add functionality to return -1 as an error value when the input argument is incorrect (negative or too large)
- Extra: make a recursive Fibonacci function (remove the for-loop)
  Note: functions calls are expensive: a for-loop is much faster than a series of (recursive) function calls

Include files

- Include files contains declarations of functions, types (structs) and constants.
- Used to speed up compilation
- Each compilation unit (`.c` file) must **`#include`** the relevant file(s)
- **`#include <stdio.h>`** for system files, **`#include "myheader.h"`** for local files

```
#ifndef FIBO_H
#define FIBO_H

long long fibonacci(int nterm);

#endif  // FIBO_H
```

Include guards: prevents a file being included multiple times

Libraries

If multiple functions will be used a lot, possibly in different projects, they can be grouped together in a library.
A library is in a sense just an object `.o` file, but with some extras.

There are two types

- static libraries: usually `.a` extension
  linked at compile time -> larger program
- dynamic libraries: usually .so (.dylib on macOS)
  linked at runtime
  You can view the relevant dynamic libraries for an executable with
  `ldd <program>` (`otool -L <program>` on macOS)

Libraries

We'll deal only with static libraries here

To create a static library, use the following commands in order:

```
  ar -rc lib<name>.a obj1.o obj2.o …
  ranlib lib<name>.a
```

(`ar` creates an *ar*chive, and `ranlib` indexes it. You can often combine the two commands into one with an extra `ar` flag:
`ar -rcs lib<name>.a obj1.o obj2.o`)

You link your program object file with the library as follows:

```
  gcc -Wall -Wextra —Wpedantic -std=c11 -c main.c
  gcc main.o -o main —l<name> -L.
```

(`gcc` acts as a *linker* in the second line. Note: no lib for the library name here)

Or in one go (compile & link):

```
  gcc -Wall -Wextra —Wpedantic -std=c11 -o main -l<name> -L.
```

Exercise

- Move the Fibonacci *function* into its own file (or move the main function into its own file): `fibo.c` and `main.c`
- Create a `fibo.h` file that has the declaration of the function. Include it in `fibo.c` and `main.c`
- Compile `fibo.c` and `main.c` to objects file (`fibo.o` and `main.o`):
  ```
  gcc -c <other flags> fibo.c
  ```
  Link the object files together into an executable
- Turn the `fibo.o` object file into a (static) library `fibo.a`
- Compile and link `main.c` with the `fibo.a` in one go
- Extra: create a `Makefile` to do all the work

Structs

A struct is a data structure to hold multiple related values together. It becomes a "new" type.

Define it in a header file (or at the top of the program, below the includes):

```
struct Pos {
  double x;
  double y;
  double z;
};
```

Then use it as follows:

```
int main()
{
  struct Pos pos2 = {.x=1, .y=-2.2, .z=0.2};
  struct Pos pos = {.x=0};
  pos.x += 5.5;
  pos2.y = pos.y;
  …
}
```

Then use it as follows in a function:

```
double calculate(struct Pos pos)
{
  double w = pos.x + pos.y - pos.z;
  return w;
}
```

Exercise

- Define a struct `Ball` that contains an `x` and `y` position, and an v_x and v_y velocity
- *Instantiate* the struct at a global level (outside `main` and other functions).
  Pick some decent initial values;
- Create a function that takes as inputs
  - a time interval `tdelta`
  - a gravity acceleration constant `g`
  - and a damping factor `f`
- and calculates the next position & velocity of the ball. The ball bounces when `y <= 0`, and loses v_y by a factor `f` (as well as reverses v_y).

- In main(), set tdelta, g and f as variables (not constants)
- Set a variable stop, which is the total time of the simulation
- Start at `t = 0`, and loop until `t > stop`
- Use `printf` to print the time, x and y position each loop iteration

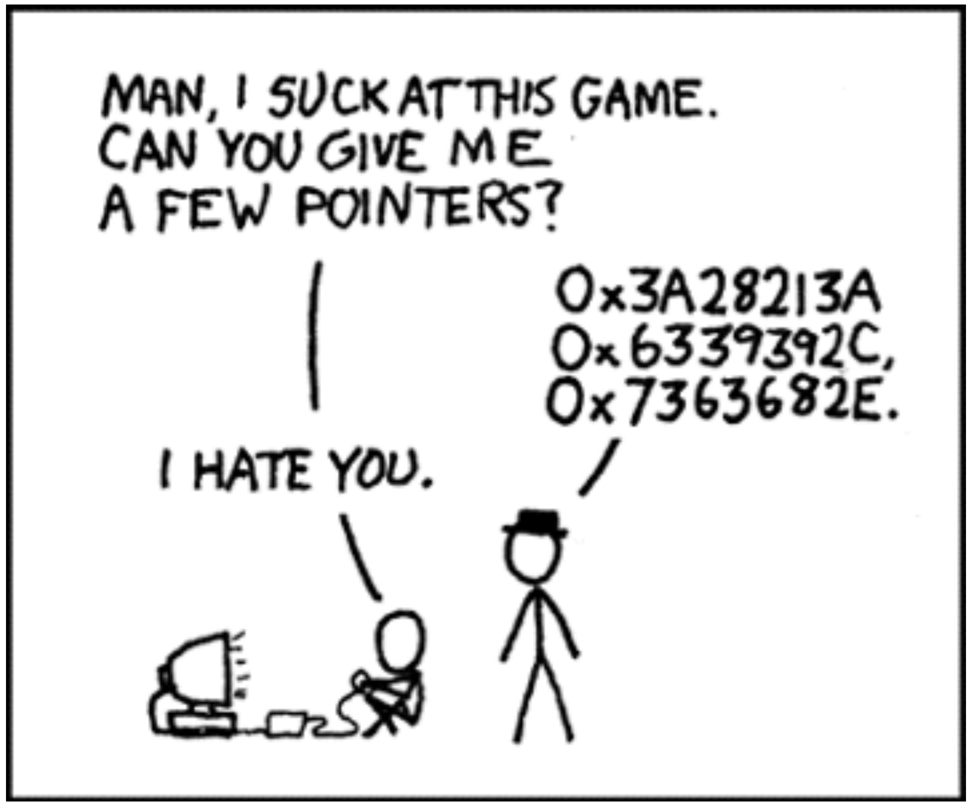
- You can redirect the output to a file: `./ball > ballpos.txt`
- You can create a plot of, for example, height versus time in any spreadsheet
- Or if you like to try gnuplot: `./gnuplot`

```
gnuplot> plot “ballpos.txt” using 1:3 with lines title “ball”
```

Pointers

Beware!

A pointer is a variable that stores the memory address of another variable: it *points* to the memory position of that other variable.


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

XKCD - Randall Munroe

```
double x = 5.5;
double y = 3.3;
double *ptr;   // pointer to double
ptr = &x;
y += *ptr;   // amounts to y += x
```

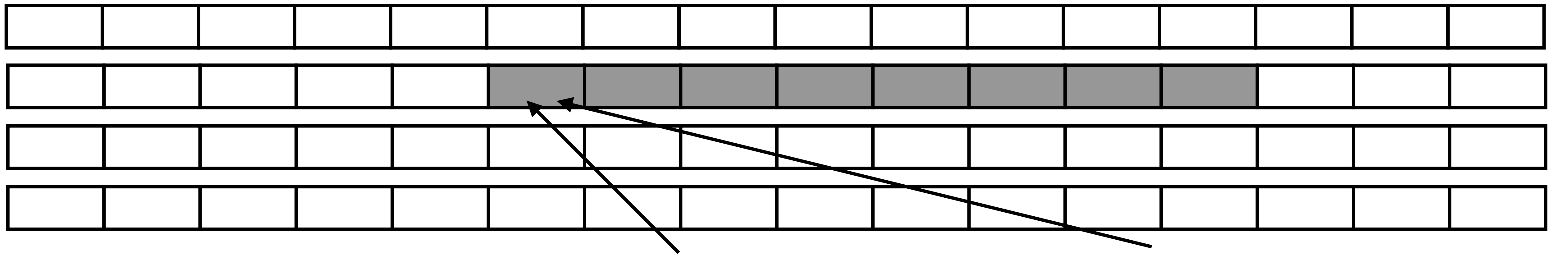
Pointers need to know about the type they point to: you can't point an **`int *ptr`** to a **`double`** (the variable size in memory is different).

Terminology:
&: address-of operator
*ptr: dereference the pointer

Pointers

Beware!

A pointer is a variable that stores the memory address of another variable: it *points* to the memory position of that other variable.



```
double a = 5.5;     double *ptr_a = &a;     double *ptr_a2 = &a;
```

I can change the variable value by assigning a new value to the *dereferenced* pointer:

```
*ptr_a += 5.5;  // a is now 11
```

But I should *not* change the pointer itself, since it will then point to random memory:

```
ptr_a = 0x3A28187E;  // NO!!!
```

Exception: NULL is a special constant, for a not-set pointer

```
ptr_a = NULL;  // ok; but don't dereference this pointer
```

Pointers - but why?

C passes arguments to functions as copy-by-value: a copy of the argument value is made. You can alter the value in the function, but you only alter the copy.

```
#include <stdio.h>
void something(double a)
{
    a = 0.1;
    return ;
}

int main()
{
    double a = 5.5;
    something(a);
    printf(“%f\n”, a);  // prints 5.5

    return 0;
}
```

Pointers - but why?

Pass a pointer to the variable, and you can alter the *contents* of the pointer (but not the pointer value itself; nor should you).

```
#include <stdio.h>
void something(double *a)
{
    *a = 0.1;
    return ;
}

int main()
{
    double a = 5.5;
    something(&a);
    printf("%f\n", a);  // prints 0.1

    return 0;
}
```

Pointers - but why?

Heed & read compiler warnings & errors

```
#include <stdio.h>
void something(double *a)
{
    a = 0.1;  // error: assigning to 'double *' from incompatible type 'double'
    return ;
}

int main()
{
    double a = 5.5;
    something(&a);
    printf("%f\n", a);  // prints 0.1

    return 0;
}
```

I forgot to dereference the pointer in the function `something`: `*a = 0.1;`

Exercise

Take the file with the ball main function, and add the following before the loop.
`scanf` reads input from the command line (and requires conversion specifiers like printf), and we can ask a user for input.
Note that `scanf` can't print output (e.g., a question string). `scanf` waits for <enter> before fully reading the input, and discards whitespace

```
int main()
{
    double g = 9.8, f = 0, tdelta = 1, stop = 1;
    printf("Value for g: ");
    scanf("%lf", &g);
    printf("Value for f: ");
    scanf("%lf", &f);
    …
    return 0;
}
```

`"%lf"` indicates a *double* explicitly.
Think "long float"

NB: `scanf` returns the number of correctly read variables. Use it to verify everything went ok.

Struct pointers

Global variables like the Ball struct are generally not a good idea
Using a pointer, we can alter the contents of a struct inside a function
Accessing members of struct pointers becomes a bit cumbersome though:

```
void calculate(struct Ball *ball, …)
{
    (*ball).x += (*ball).v_x * tdelta;
    …
}
```

There is a shortcut for that, the arrow operator:

```
void calculate(struct Ball *ball, …)
{
    ball->x += ball->v_x * tdelta;
    …
}
```

Exercise

Move the global `ball` variable into the `main` function.

Change the function that calculates the ball's movement to accept a pointer to a `Ball` struct (as well as the usual g, f, tdelta arguments).

Don't forget to change the call to the function in `main` as well.

Arrays

Beware!

Pointers have more uses: for arrays whose size is not known at compile time.

Compile time array (these can never be very large):

```
double a[50];
double b[] = {1, 2, 3, 4, 5};
double c[1048576];  // maximum on my system
```

With unknown input, or array sizes larger than the maximum, you need to *allocate* a block of memory, and *point* a variable to the start of that block.

```
#include <stdlib.h>
double *a = malloc(n * sizeof *a);
```

When you're done with the array, you need to *free* the memory (de-allocation):

```
free(a);
```

Now a is a pointer that points to a region in memory where no memory is reserved, and *a can become anything. Set it to NULL to be safe:

```
a = NULL;
```

Arrays

What can go wrong?

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main()
{
    long long n = 0;
    printf("Array size: ");
    scanf("%lld", &n);
    double *a = malloc(n * sizeof *a);
    for (long long i = 0; i < n; i++) {
        a[i] = sqrt(i);
    }
    free(a);

    return EXIT_SUCCESS;
}
```

Failing to allocate memory
(I should check for `a == NULL`)

Out of bounds (`i >= n`)

Forget to free (memory leak)

Free an already freed pointer
(double free)

Use `a` again after free.

Important: arrays in C are 0-based.

Arrays & strings

A string is an array of char’s, with a final `‘\0’` (`NUL`) character.
The memory size of the string is the number of characters, including the `NUL` character
NB: `NUL != NULL`

```
char varstring[] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
```

But better written as a pointer to a character.
Note the *double* quotes around `hello`.

```
char *string = “hello”;
```

While not written, the above string has an implicit `‘\0’` at the end

Exercise

Store the results of the ball calculation in three arrays: `timestamps`, `xpos` and `ypos`. You have to calculate the array size from the input time parameters, and dynamically allocate the array (and free the allocation afterwards).

To convert a double to an int, you *cast* it:

```
int a = (int) 5.0;
```

Be aware that this *truncates* the result (that is, rounds down). For example:

```
int a = (int) (10.0/3.0);
a == 3
```

Move the `printf` function out the main loop, and create a second loop afterwards where you *only* print the results.
(Normally, the calculations would all be done in one function, and printing of the results in another function.)

Converting to C++

Most C that we used so far it also valid in C++.
But quite a number of things are not used in C++ this way.
We'll change our current ball program to a C++ program step by step.

- File extension: `.cpp`, `.cxx` or `.cc`. Don't use capital `.C`
- Local (user) header files: `.hpp`
- System header files don't have an extension. For example, `<iostream>`, `<vector>`
- C headers are replaced by removing the extension, and prepending "c": `<cstdio>`, `<cmath>`, `<cstdlib>`
- Compiler: `g++` or `clang++`.
- Compiler standard flag: `-std=c++14` or `-std=c++11`
  (or even `-std=c++17` or `-std=c++1z`)

Exercise: change the ball program and compile it with a C++ compiler.

You'll run into a few incompatibilities…

Converting to C++

Cast the return value of malloc in C++:

```
double *timestamps = (double *) malloc(n * sizeof *timestamps);
```

While you won't get a warning doing this in C, don't do it: it hides a potential error (forgotten `#include <stdlib.h>`).
This is why C++ != C with some extras

C++ does not have designated initializers for structs. Instead, use

```
struct Ball ball = {0, 10, 1, 0};
```

You can do this as well in C, but it is less clear than using the member names.
In C++, structs are quite different beasts, and initialization like this is not really used.

std::vector

std::vector is a much more flexible array type

```
#include <cstdio>
#include <vector>

int main()
{
    std::vector<double> a;
    printf("%lu\n", a.size());
    a.push_back(4.4);
    a.push_back(2.2);
    printf("%lu\n", a.size());
    printf("%f\n", a[1]);
}
```

`std::` namespace of the standard library

Takes the type as a *template* parameter

No need to set the size beforehand

Just add elements on the fly

Keeps track of its own size (`.size()` is a *member* function)

We can still access elements normally

An array *size* is unsigned long `"%lu" (size_t)`

No need for `free` at the end

Exercise: replace malloc/free with `std::vector`

std::map

std::map is the C++ stdlib dictionary or hash

```
#include <cstdio>
#include <map>
#include <string>

int main()
{
    std::map<std::string, double> map;
    map[“hello”] = 5.5;
    map[“world”] = 42;
    printf(“%lu\n”, map.size());
}
```

Namespaces

`<namespace>::<element>`

Can be tedious to type and cumbersome to read:

`std::map<std::string, std::vector<double>> strings;`

but avoids problems with multiple libraries having identically named functions

Shortcuts:
(at top level, usually after imports)

`using namespace std;`	Brings *everything* (from the include files) into the main namespace Convenient in short programs
`using std::vector;`	Brings only the mentioned functions, types, classes etc from the relevant include files into the main namespace Better, but still not as explicit as `std::vector`

You can always use `std::vector` anyway.

Items in the main namespace can be accessed using just the double colon
`::<item>`

Functions in C++

Functions in C++ can be overloaded: multiple functions can have the same name, but different arguments. (NB: the return-type cannot be used for overloading.)
The compiler figures out which one to call when

```
int f(int n)
{
      return n - 10;
}

double f(double x)
{
      return x + 10;
}
printf(“%d”, f(5));  // -5
printf(“%f”, f(5.0));  // 15
```

If `int f(int n)` is not defined, `f(5)` would implicitly cast 5 to 5.0 and call `double f(double x)`.
This only works from integer to floating point, not the other way around
Math functions are overloaded (for float / double / long double)

In C11: type-generic expressions

Functions in C++

Function arguments can have default values, from right to left.
You can’t name arguments though (like in Python)

```
int f(int n, int sub=10)
{
      return n - sub;
}

printf(“%d”, f(5));  // -5
printf(“%f”, f(5, 5));  // 0
printf(“%f”, f(5, sub=5));  // error
```

Note: default arguments can’t be used to distinguish between overloaded functions:

```
int f(int n)
{ … }

int f(int n, int sub=10)
{ … }
```

Compilation error

input & output: iostream

C++ uses functionality from the <iostream> header

```
std::cout << 5.5 << ", " << 123 << std::endl;
```

- `std::cout` is not an overloaded function (it's a class), but the << operator is overloaded. Think of it as sending things *into* c-out (console output).
- Each type defines how it is printed with `std::cout`, and `std::cout` uses that. No more specific conversion specifiers.
- `std::endl` replaces `"\n"`: it is a more generic newline

The opposite is std::cin, for reading input from the console

```
std::cint >> myvar;
```

- `std::cint` knows about the type of `myvar`, and will convert the input accordingly.
- No pointer needed.

Exercise: replace `printf`/`scanf` with `std::cout`/`std::cin`

References

C uses pointers, among others for when function arguments need to be modified
C++ also has references, in addition to pointers:

```
int i = 5;
int& ref = i;
ref = 6;  // i == 6

int *ref = &i;
*ref = 7;  // i == 7, ref == 7


void f(int& i)
{
    i = 15;
}
```

References act like an alias
They make code cleaner & safer
As with pointers, they can’t function on their own
Generally prefer references over pointers

Exercise: replace the Ball struct pointer with a reference

Pointers in C++

If C++ uses references, when do you need pointers?

If you want to create new objects, arrays and such dynamically (not knowing the size in advance).
Though better solutions like std::vector may exist.

C++ doesn't use malloc and free functions. It has the operators new and delete

```
double *a = new double;
…
delete a;
```

The above is pretty silly, but you can use it with structs (and classes) as well:

```
Pos *p = new Pos;
…
delete p;
```

Note: the struct keyword has gone: Pos now functions as a fully new type

(For arrays, there exist `new[]` and `delete[]`. Again, better to use `std::vector`)

Pointers in C++

C++11 has a unique pointer type
Such a pointer is automatically deleted (de-allocated) when it goes *out of scope*
This avoids memory leaks

```
#include <memory>

{
    std::unique_ptr<Pos> p(new Pos);

    // no delete, despite new earlier
}
```

Note: In C++14 there is `std::make_unique<Pos>()`, which is safer in some contexts

Constants in C++

`const` exists in C++ and C, but it is better defined in C++

`const` exist in the current scope only: you can set a constant for a function only.

```
void f(int a)
{
    const float my_pi = 3.14;
    …
}
```

Constants have a type, for safety.

Use them in C++ instead of `#define`

auto keyword in C++

`auto` exists in C++ and C, but its meaning has changed.
No use (anymore) it in C

If a type can be inferred by the compiler, you can use auto.
This is convenient when you have elaborate types, or in a for loop with iretaros (not discussed here)
`const` exist in the current scope only: you can set a constant for a function only.

```
auto a = 1;  // compiler infers int
auto b = 5.5;  // compiler infers a float
auto c = std::string(“hello”);  // compilers infers a std::string
```

character strings are turned into std::string,
because `strings` is the type of strings

```
std::vector<std::string> strings = {“hello”, “world”};

for (auto& s : strings) {
    std::cout << s << std::endl;
}
```

This is a C++11 range-based for loop
You iterate over the actual values
No indices are used

`s` is a *reference* of type `std::string` (reference saves copying)

<templates>

Templates are used when you create generic functions, or container types

```
template <typename T>
T max(T a, T b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

This defines the function max that has two arguments to the same type T, and also returns the type T.
The compiler will make sure the two arguments are of the same type.
You can compare two `std::strings` this way, but if you use

```
max(5, 5.5)
```

you'll get an error.
There is no implicit cast to a double!

There are other uses for templates as well, with classes that can use generic types.
This is where `std::vector<T>` comes into play, or `std::unique_ptr<T>`

classes

C++ structs can also contain functions, next to variables

C++ has a new keyword as well: `class`

The main difference between a class and a struct is that
 in a struct, everything is `public` by default
 in a class, everything is `private` by default

`public` and `private` basically mean:
If you're using a class or struct from a library:
 you can't access anything that is declared `private`
 you can access everything that is `public`

To confuse things further, things can also be `protected`. We won't deal with that here

The convention tends to be:
 use a class when you use member functions (functionality)
 use a struct when you only use member variables (data)
 declare private and public explicitly

classes

```
class Ball
{
public:
    void calculate(double tdelta, double g=9.8, double f=1.0);
private:
    double m_x, m_y;
    double m_vx, m_vy;
};
```

`m_` is a style convention; up to you really

```
void Ball::calculate(double g, double f, double tdelta)
{
    <calculation>

    <you can access m_x, m_y, m_vx and m_vy directly>
    <inside a member function>
}
```

In our main loop, we can create a `Ball` instance and access `ball.calculate`, but not `ball.m_x` and `ball.m_y`

classes

```
class Ball
{
public:
    void calculate(double tdelta, double g=9.8, double f=1.0);
    double x() { return m_x }
    double y() { return m_y }
private:
    double m_x, m_y;
    double m_vx, m_vy;
};
```

These are getter functions
Note how they are declared in-line
There are no getter functions for the velocity; these remain hidden

Sample usage:

```
Ball ball;

ball.calculate(1.5);
std::cout << ball.x() << ", " << ball.y() << std::endl;
```

classes

```
class Ball
{
public:
    void calculate(double tdelta, double g=9.8, double f=1.0);
    double x() { return m_x }
    double y() { return m_y }
private:
    double m_x, m_y;
    double m_vx, m_vy;
};
```

These are getter functions
Note how they are declared in-line
There are no getter functions for the velocity; these remain hidden

Getters prevents you from accidentally changing `m_x` outside `calculate()`
If you want that, you'd need to define a setter.
A setter could test for physical values first before setting a member variable

This is a good thing for a large framework that lots of people use.
For a small project, you may be perfectly fine with public member variables
Overall, it shouldn't matter too much

constructors

What about the initial values of m_x, m_y etc?
You can set those in a constructor of a class

A constructor is a special member function:

- same name as the class
- no return type
- can have arguments (with default values)
- can be overloaded (same for any member function)
  that is, multiple constructors can exist (this goes for any member function)
  Which constructor is used, depends on how the class is instantiated

There are also destructors

- same name as the class, with ~ prepended (not operator)
- called when a variable goes out of scope (dies)
- no return type
- cannot have arguments
- cannot be overloaded

Useful to deallocate memory, close a file or release a lock
Won’t be using it here
If not implemented, a default (“do nothing”) destructor is used

constructors

```
class Ball
{
public:
    Ball(double gravity=9.8, double efficiency=1);
    …
private:
    double gravity, efficiency;
};

Ball::Ball(double gravity, double efficiency)
{
    this->gravity = gravity;
    this->efficiency = efficiency;
}

int main()
{
    Ball ball{9.8, 0.5};
    Ball ball2;
    …
}
```

Curly braces are C++11
(uniform initialization syntax)
Before that, () was used, but this
can be confused with a function call.
Parentheses are certainly still allowed

constructors

```
Ball::Ball(double gravity, double efficiency)
{
    this->gravity = gravity;
    this->efficiency = efficiency;
}
```

This is so common, that a different syntax is usually used:

```
Ball::Ball(double gravity, double efficiency) :
    gravity(gravity), efficiency(efficiency)
{
}
```

In this case, you could read this similar to

```
double gravity = 9.8;
double gravity2(gravity);
double gravity2{gravity};  // C++11 alternative
```

It guarantees a constructor call for the member variables, even for simple types

constructors

Such simple constructors are often directly implemented in the class definition

```
class Ball
{
public:
    Ball(double gravity=9.8, double efficiency=1) :
        gravity(gravity), efficiency(efficiency) { }
    …
private:
    double gravity, efficiency;
    …
};
```

Exercise

Implement the ball calculation inside the Ball struct

- use `class` instead of `struct`
- your choice of `private` / `public`
- your choice of member variable naming style
- gravity g and efficiency f should become member variables as well

The constructor

- gravity and efficiency should be mandatory arguments
  The position and velocity variables should be default arguments

Extra

- Store the positions and time in a vector, as member variables of the class
- Add a member function that takes care of the full loop: it should take a start and stop time, and a timestep. It should call the calculation function each iteration.
  Thus, everything we did in `main()`, is now done inside the class (apart from the output)
- Since you're now not calling the calculation directly, you could make it private.
- You'll need to make the relevant output variables public, or define a getter function, if you want to be able to read and print the final results
  Or: you could define a `print()` or `output()` class member function for this.

What else is there

Operator overloading

```
std::string(“hello”) + std::string(“world”)
auto ball = ball1 + ball2
```

Much more in the standard library. Or rather, Standard Template Library, STL:

```
std::find, std::sort, std::set
```

If you’re looking for functionality for your program, first see if it exists in the STL

Exceptions

Command line arguments

Mixing C and C++ code

(File) I/O

Inheritance

…

Don’t go overboard

C++ allows for a lot of options

Don’t use classes where you don’t really need them

Keep your code clean and simple (KISS)

Often, using C++ in a C-style manner (but without the dangers of pointers) works pretty well, in particular for scientific software

More: particle physics simulation

DIY http://home.thep.lu.se/~torbjorn/pdfdoc/worksheet8200.pdf

From previous years, by Peter Skands