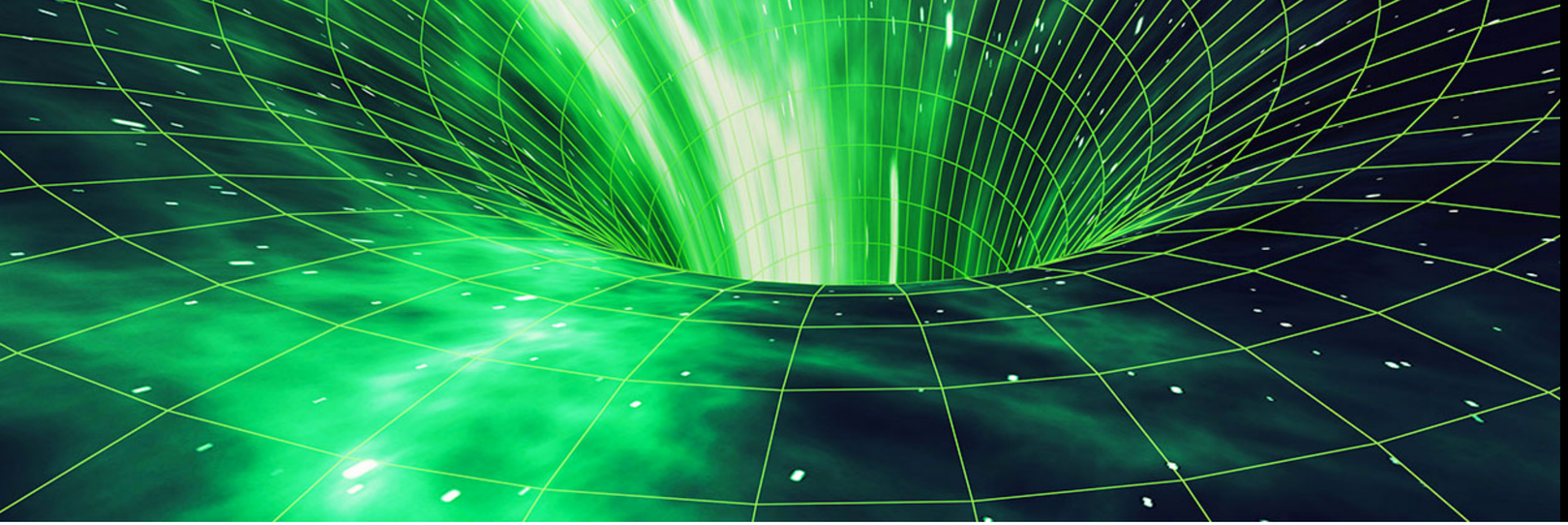
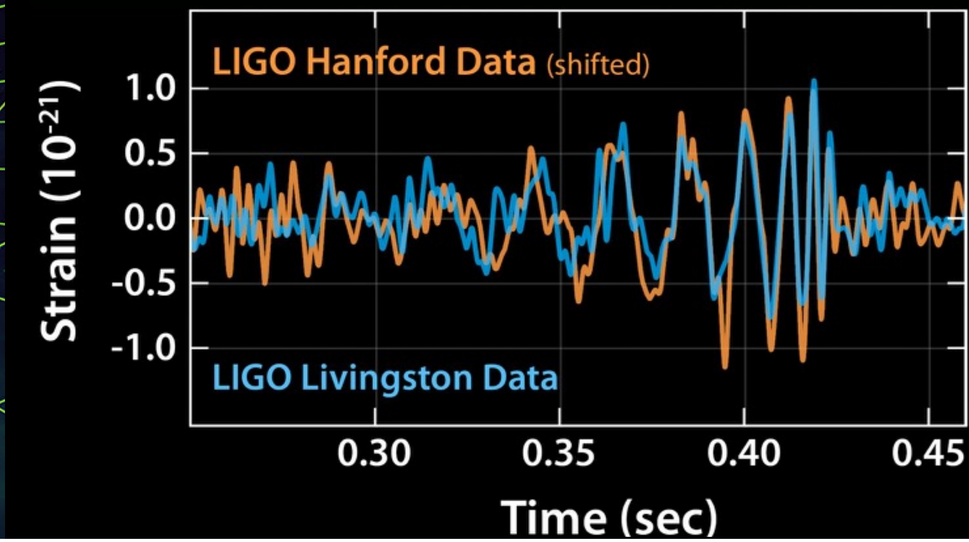
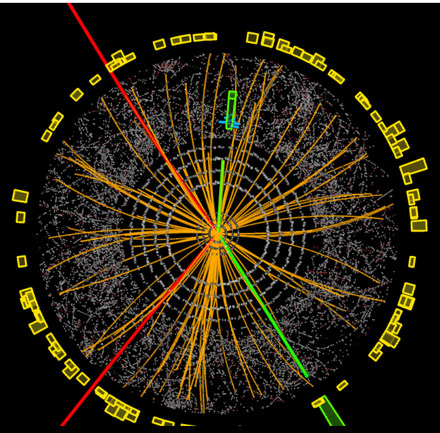




Computing Methods in Physics 1*

***i.e., “data analysis oriented”**



Introduction

About this Course

Who? When? Where?

Francesco Pannarale

- francesco.pannarale@uniroma1.it
- https://sites.google.com/uniroma1.it/francesco-pannarale-eng

} Any underlined text is a link

- "Marconi", 2nd floor, room 214
- Office hours: Wednesdays 10:00-12:00 (email me first!)

Lectures

- Throughout the semester: Tuesdays 8-10 and Thursdays 12-14 (Aula Majorana)
- Occasionally: Mondays 8-11 (Sala Calcolo, CU033) → more on this later later
- Google classroom: xan5ire (register!)

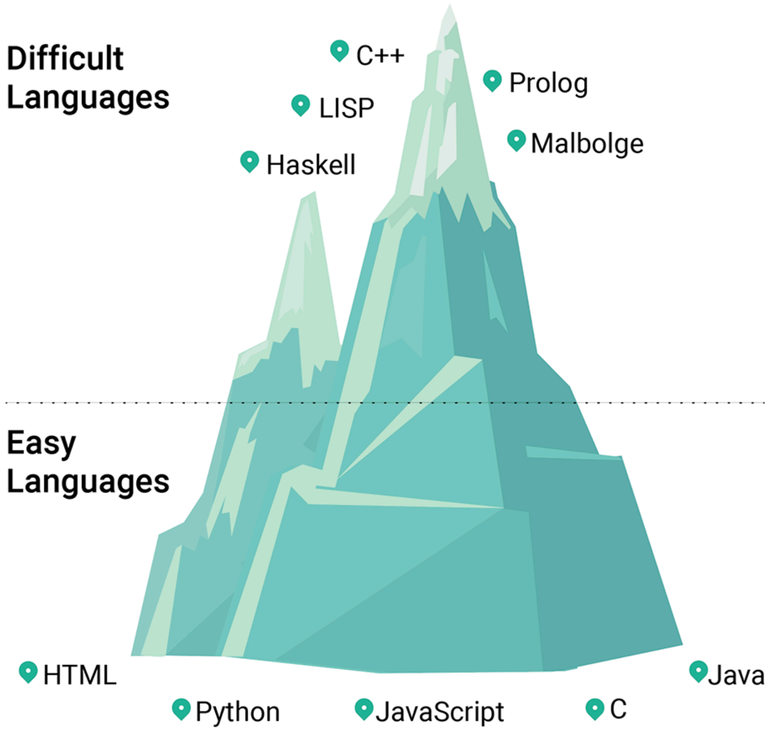
About this Course

What?

- This specific flavour of Computing Methods for Physics aims at providing you with and introduction to tools currently used in data analysis

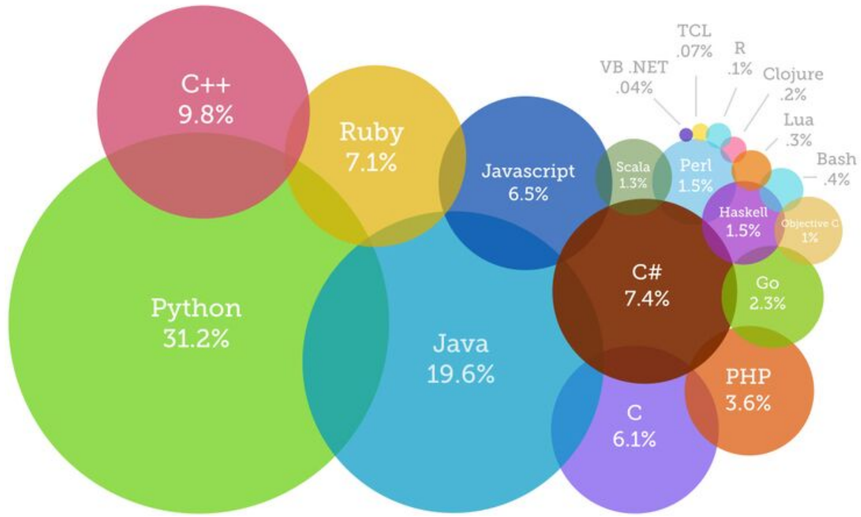
- 50% C++ (object-oriented programming and `ROOT`) + 50% Python (`NumPy`, `SciPy`, `Matplotlib`)

- Other important ingredients
  1. Collaborative coding & version control (`git`)
  2. Debugging
  3. Optimization



About this Course

Why?



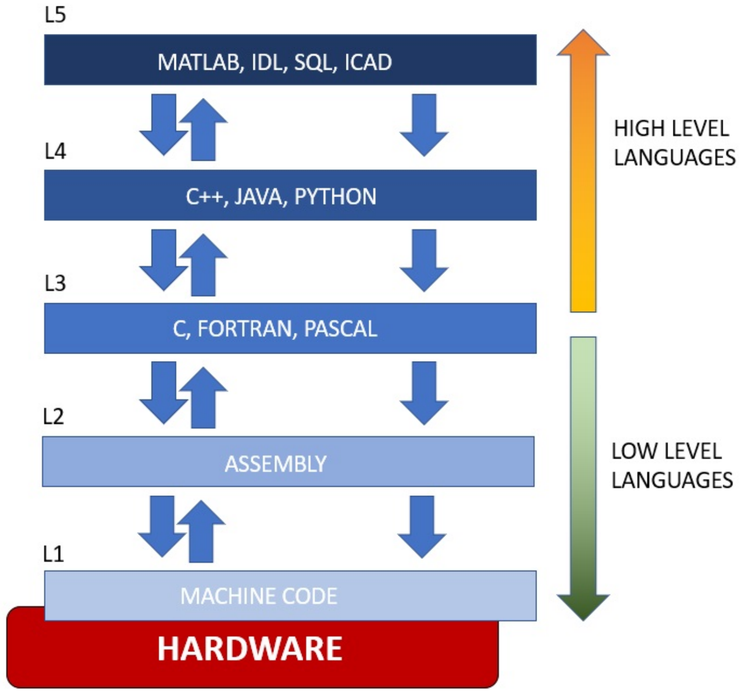
You will need to work on a thesis and find a job

C++

- Primary programming language in several fields of physics, including high-energy physics
- Benefits of C: modular, efficient, close to the machine, portable
- Supports object-oriented programming concepts (data abstraction, inheritance, polymorphism)

Python

- Very popular and versatile language in both industry and physics
- Can be used interactively
- Countless open-source codes, free tutorials, etc.



About this Course

Disclaimer

- I am a gravitational-wave physicist
- I am not a software engineer, a computer scientist, a C++/Python guru, etc.
- I have coded and dealt with codes (two different things!) daily for the last ~17 years in various areas of my field, in several languages, in teams with $O(1)$ to $O(100)$ members, focusing on high-performance and high-throughput computing, etc.
- This course will not cover everything your careers will throw at you, but it will prepare you to tackle programming and computing challenges
- We will not learn/use Java, which also has a high demand in the private sector

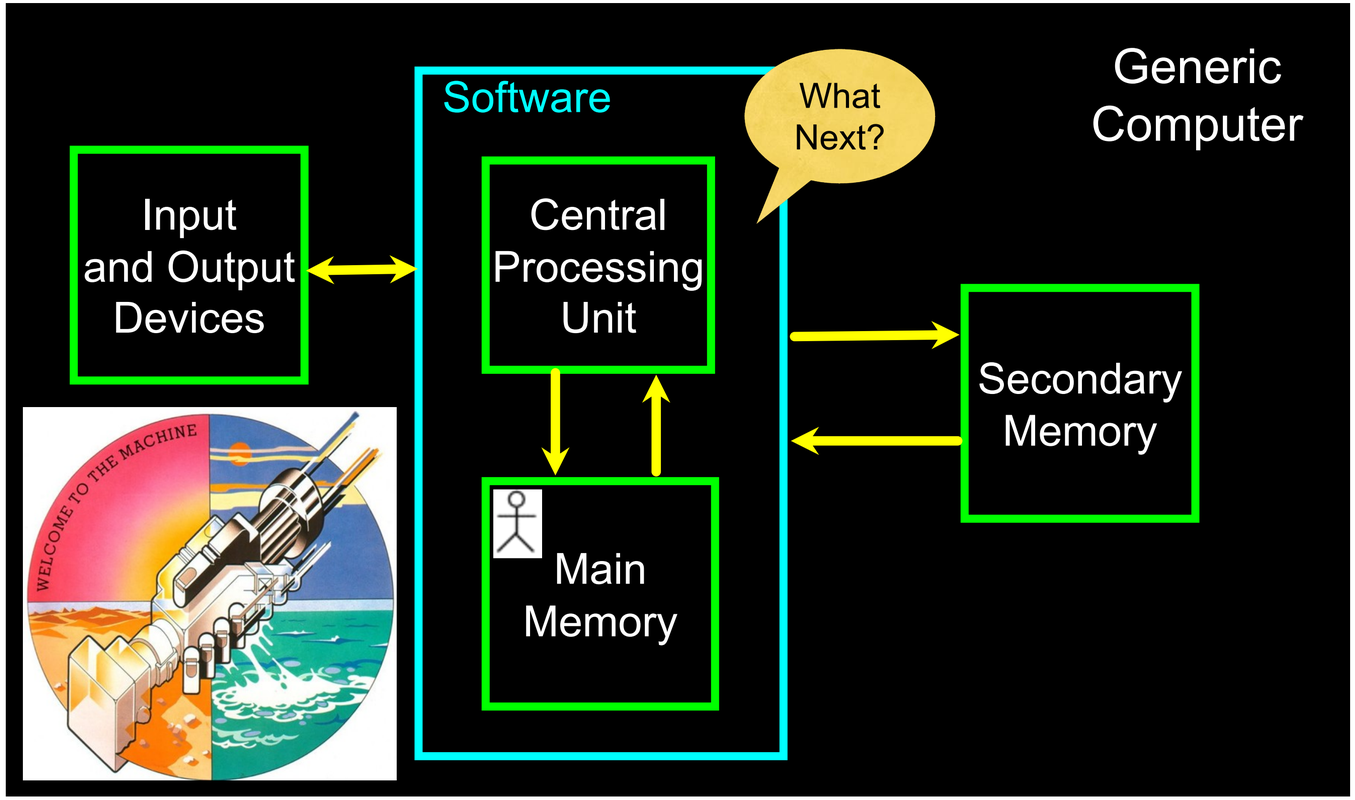
About this Course

Admin

- I will share slides and examples we discuss
- Mondays from October ~10 are dedicated to 3 hour assisted lab sessions
  - You will be working in pairs
  - You are encouraged to use GitHub and Slack

- Exams
  - Dates available on InfoStud
  - You will receive a PDF file with a C++ problem and a Python problem
  - You will have 4 hours to submit a zip file with your project
  - An oral discussion about your code and coding choices will follow
  - Your final grade depends on the quality of the code you submit and on the discussion

Programming and You



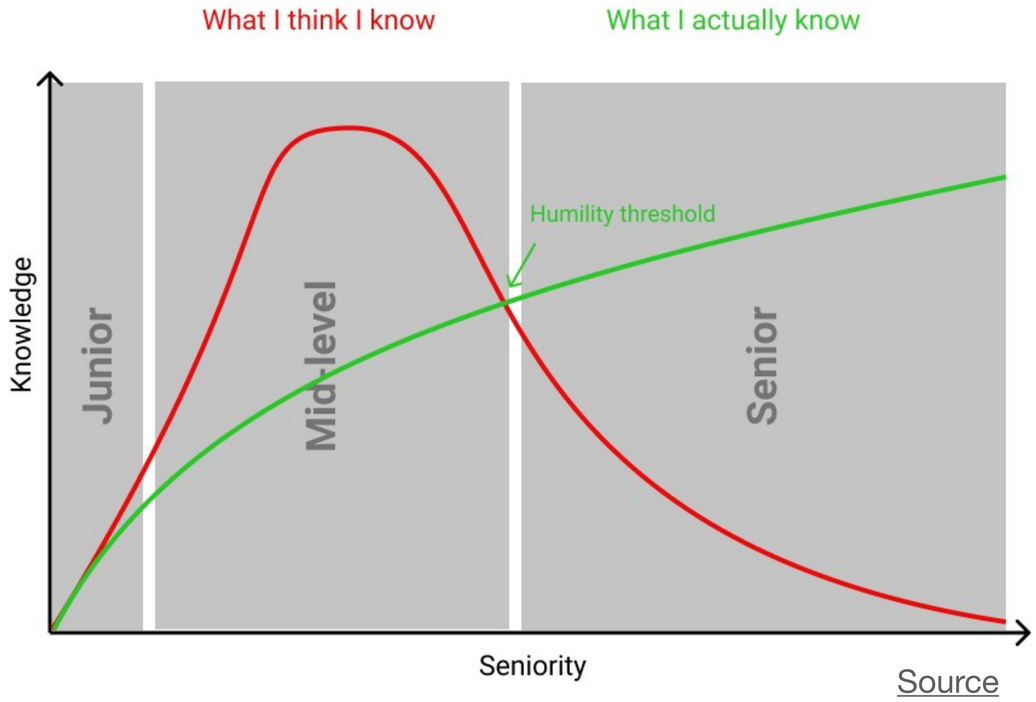
- Computers are built for one purpose: to do things for us (e.g., physics calculations)
- CPUs are dumb but fast
- Use programming languages to write the sequence of stored instructions (code/program/software) that are necessary for the computer to do what you need it to do
- By programming, you place pieces of your intelligence in the machine

You and Programming

- Learning languages requires you to try them out: you will be hitting failures and you will be debugging over and over… so practice, practice, practice!

- *If you are teaching today what you were teaching five years ago, either the field is dead or you are.*

  Noam Chomsky



Source

History of C++

History of C++

Prehistory

- Early on, programmers worked with the most primitive computer instructions: they provided instructions with long strings of 0's and 1's (machine language)
- Assemblers were invented to map machine instructions to human-readable and manageable mnemonics (e.g., `ADD` and `MOV`)
  - Bug prone, redundant, hard to write/read
  - Requires detailed knowledge of hardware
  - Not portable

```
    .equ    LAST_RAM_WORD,   0x007FFFFC
    .equ    JTAG_UART_BASE,  0x10001000
    .equ    DATA_OFFSET,     0
    .equ    STATUS_OFFSET,   4
    .equ    WSPACE_MASK,     0xFFFF

    .text
    .global _start
    .org    0x00000000

_start:
    movia       sp, LAST_RAM_WORD
    movi        r2, '\n'
    call        PrintChar
    movia       r2, MSG
    call        PrintString
_end:
    br          _end

PrintChar:
    subi        sp, sp, 8
    stw         r3, 4(sp)
    stw         r4, 0(sp)
    movia       r3, JTAG_UART_BASE
pc_loop:
    ldwio       r4, STATUS_OFFSET(r3)
    andhi       r4, r4, WSPACE_MASK
    beq         r4, r0, pc_loop
    stwio       r2, DATA_OFFSET(r3)
    ldw         r3, 4(sp)
    ldw         r4, 0(sp)
    addi        sp, sp, 8
    ret

PrintString:
    subi        sp, sp, 12
    stw         ra, 8(sp)
    stw         r3, 4(sp)
    stw         r2, 0(sp)
    mov         r3, r2
ps_loop:
    ldb         r2, 0(r3)
    beq         r2, r0, end_ps_loop
    call        PrintChar
    addi        r3, r3, 1
    br          ps_loop
end_ps_loop:
    ldw         ra, 8(sp)
    ldw         r3, 4(sp)
    ldw         r2, 0(sp)
    addi        sp, sp, 12
    ret

        .org        0x1000
MSG:    .asciz      "Hello World\n"
        .end
```

History of C++

Ancient History

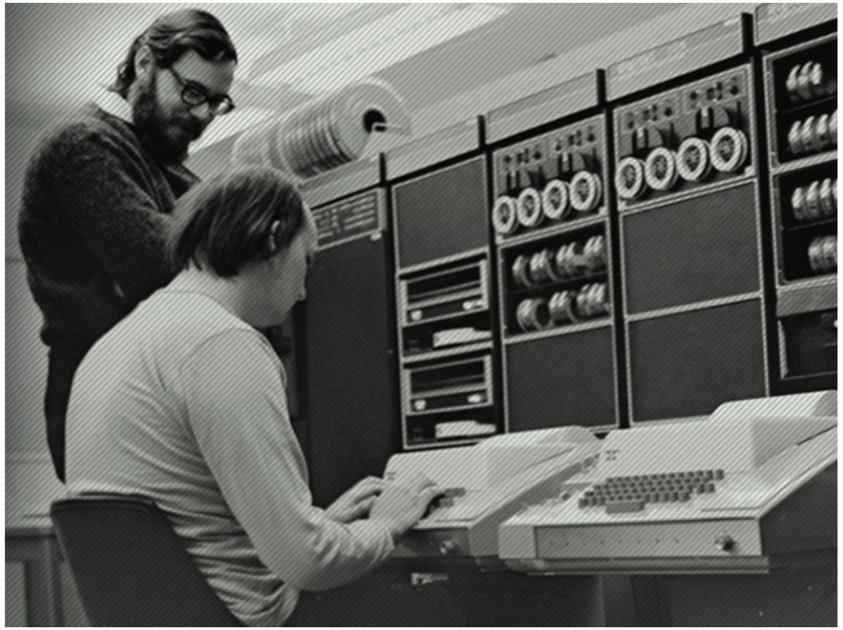
- Higher-level languages (BASIC, COBOL) evolved to allow people to work with something approximating words and sentences (e.g., `Let I = 100`).
- These instructions were then translated into machine language by:
  - Interpreters – translate and execute programs on the spot as they read them
  - Compilers – translate (compile) source code into an intermediary form (object file) and then invoke a linker, which combines the object files into an executable program

Compiled programs (e.g., C) run very fast because the time-consuming task of translating the source code into machine language has already been done at compile time, and it is not required when executing the program. Unlike interpreters, they do not need to be distributed along with the code.

History of C++
Middle Ages

- In 1969, Dennis M. Ritchie and Ken Thompson (AT&T, Bell Labs) start developing UNIX, a new operating system for a large computer that could be used by thousands of users

- It was written in assembly code, but – other than assembly and FORTRAN (FORmula TRANslator) – it had an interpreter for the programming language B, a high-level language also developed at Bell Labs



History of C++

Modern History

- B lacked data-types and structures: Ritchie set off to develop C, an evolution of B
- C became very popular and was ported to a variety of hardware platforms; it was standardized in 1990 by the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI)
- Some applications
  - Operating systems and compilers
  - Computer games
  - Special effects for Star Wars

History of C++
Contemporary History

- The C++ programming language was created by Bjarne Stroustrup and his team (AT&T, Bell Labs) to help implement simulation projects in a way that is object-oriented as well as efficient and portable (inheriting from C)

- The earliest versions (1980s) were referred to as “C with classes” (++ is the C increment operator)

- “C++ is for people who want to use hardware very well and manage the complexity of doing that through abstraction.”

- “C++ is not for everybody. It is generated to be a sharp and effective tool […] definitely for people who aim at some kind of precision.”

- “C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it plows your whole leg off.”

Introduction to C++

What is “Object-Oriented Programming?”

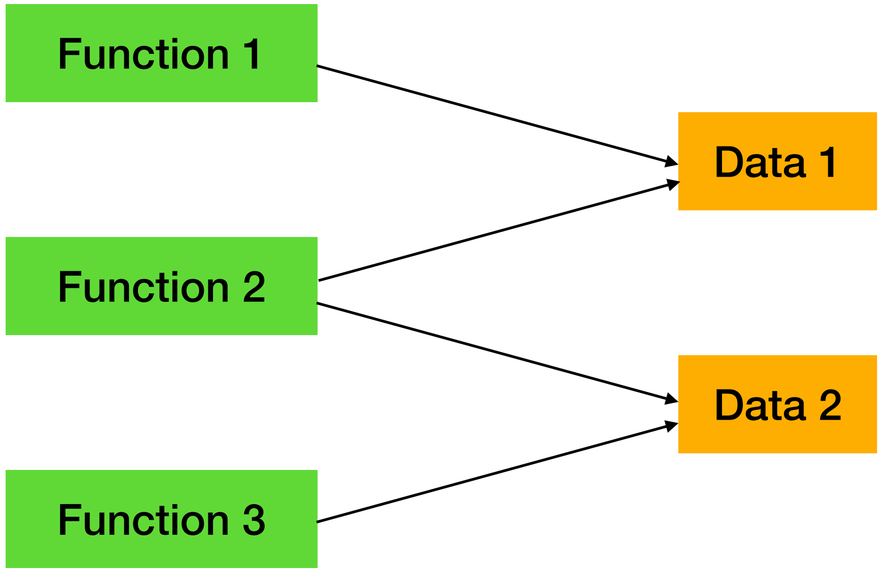
Definitions

- Objects are software units modelled after entities in real life
  - They have attributes: e.g., length, density, color
  - They have a behaviour and provide functionalities: e.g., a door can be opened/shut, a nucleus can decay
- Object-oriented programming means writing code in terms of objects, i.e., well defined units which have attributes and offer functionalities
  - A program hence consists of interactions between objects using methods offered by each one of them
  - Objects are “smart” data structures: they are data with behaviour

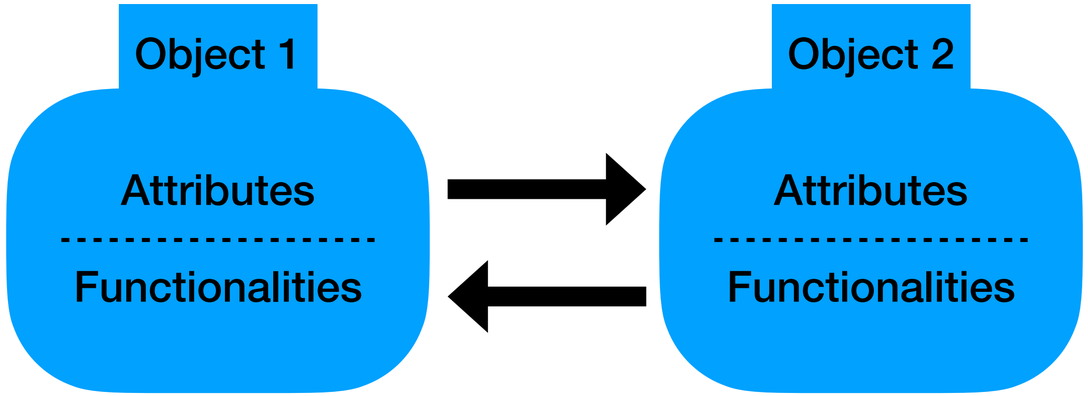
What is “Object-Oriented Programming?”

In Pictures

Traditional concept



Object-oriented concept



Characteristics of C++

From object-oriented programming

- Data abstraction – the creation of classes to describe objects
- Data encapsulation for controlled access to object data
- Inheritance by creating derived classes (including multiple derived classes)
- Polymorphism – the implementation of instructions that can have varying effects during program execution

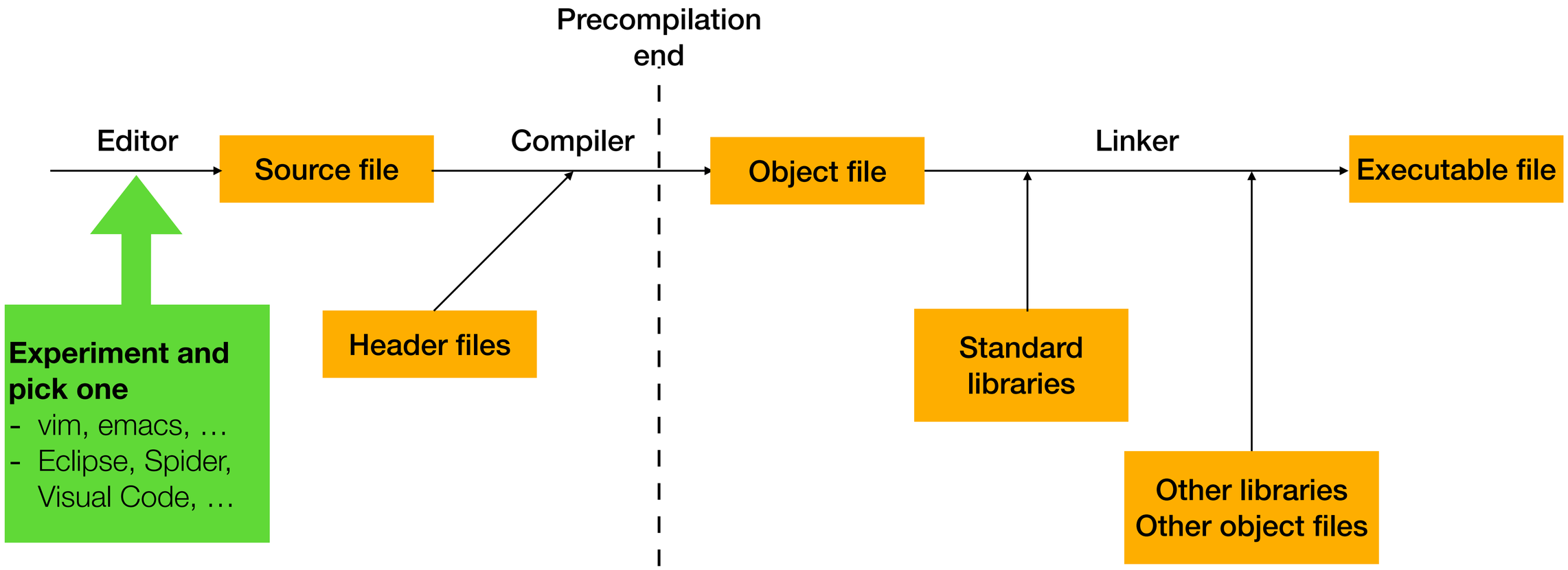
From C

- Universal
- Efficient
- Close to the machine
- Portable

Extensions

- Templates – allow the construction of functions and classes based on types not yet stated
- Exception handling

Translating a C++ Code



First Lines of C++

Based on `examples/01/Welcome.cpp`

CODE LINES

First Lines of C++

Based on `examples/01/Welcome.cpp`

```
// Your first C++ application!
#include <iostream> // Required to perform C++ stream I/O

/* Function main begins
   program execution */
int main() {
   return 0; // Indicate that program ended successfully
} // End function main
```

All statements must end with a semi-colon (`;`)

- Comments run from `//` to the remainder of the line or are enclosed between `/*` and `*/`
- Preprocessor/precompiler instructions: include C++ stream I/O (unnecessary here) before compiling
- Any C++ application **must have** a `main` method, that **must return** an `int`
  - Return value can be used by user/client/environment, e.g., for error handling

First Lines of C++

Based on `examples/01/Welcome.cpp`

```
// Your first C++ application!
#include <iostream> // Required to perform C++ stream I/O

/* Function main begins
   program execution */
int main() {
   return 0; // Indicate that program ended successfully
} // End function main
```

- With `include` the preprocessor replaces the user directives with the requested source code
- The syntax `#include "Foo.h"` is for header files that are not part of the standard C++ library
- To see this happening, look at your precompiled code by running:

  ```
  g++ -E Welcome.cpp
  ```

First Lines of C++

Based on `examples/01/Welcome.cpp`

```
// Your first C++ application!
#include <iostream> // Required to perform C++ stream I/O

/* Function main begins
   program execution */
int main() {
   return 0; // Indicate that program ended successfully
} // End function main
```

- Compile `Welcome.cpp` (into a binary output called `Welcome`) and run (the binary output)

```
g++ -o Welcome Welcome.cpp
./Welcome
```

"TRY IT YOURSELF" MATERIAL!

First Lines of C++

Based on `examples/01/Welcome.cpp`

```
// Your first C++ application!
#include <iostream> // Required to perform C++ stream I/O

/* Function main begins
   program execution */
int main() {
   return 0; // Indicate that program ended successfully
} // End function main
```

- Compile `Welcome.cpp` (into a binary output called `Welcome`) and run (the binary output)

```
g++ -o Welcome Welcome.cpp
./Welcome
```

Experiment again by: including `Foo.h`, using `void main`, not returning an `int`

Output with `iostream`

Based on `examples/01/SimpleOutput1.cpp`

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

int main() { // main begins here

   // print message to STDOUT
   cout << "Moving baby steps in C++!" << endl;

   return 0;

} // end of main
```

- `iostream` provides output capabilities to your program
- `cout`: character output
- `endl`: end of line, starts a new line
- Both are defined in the standard (`std`) `namespace` [more on this later]

`Shell input and output`

Output with `iostream`

Based on `examples/01/SimpleOutput1.cpp`

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

int main() { // main begins here

   // print message to STDOUT
   cout << "Moving baby steps in C++!" << endl;

   return 0;

} // end of main
```

- `iostream` provides output capabilities to your program
- `cout`: character output
- `endl`: end of line, starts a new line
- Both are defined in the standard (`std`) `namespace` [more on this later]

```
$ g++ -o SimpleOutput1 SimpleOutput1.cpp
$ ./SimpleOutput1
Moving baby steps in C++!
```

Output with `iostream`

Based on `examples/01/SimpleOutput2.cpp`

```
#include <iostream>


int main() { // main begins here

   // print message to STDOUT
   std::cout << "Moving baby steps in C++!" << std::endl;

   return 0;

} // end of main
```

- Alternative version without the the standard (`std`) namespace
- Shows you why they are convenient

```
$ g++ -o SimpleOutput2 SimpleOutput2.cpp
$ ./SimpleOutput2
Moving baby steps in C++!
```

First C++ Compilation Errors

Based on `examples/01/BadCode1.cpp`

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

int main() { // main begins here
   int nIterations;
   cout << "How many
           iterations? "; // cannot break in the middle of the string!
   cin >> nIteration; // wrong name! the s at the end missing
   // print message to STDOUT
   cout << "Number of requested iterations: " << nIterations << endl;

   return 0 // ; is missing!
} // end of main
```

`iostream` provides input capabilities via `cin >>`

First C++ Compilation Errors

Based on `examples/01/BadCode1.cpp`

- See the compilers response to this!

  `g++ -o oh_no BadCode1.cpp`

```
BadCode1.cpp:9:12: warning: missing terminating '"' character [-Winvalid-pp-token]
   cout << "How many
           ^
BadCode1.cpp:9:12: error: expected expression
BadCode1.cpp:10:24: warning: missing terminating '"' character [-Winvalid-pp-token]
           iterations? "; // cannot break in the middle of the string!
                       ^
BadCode1.cpp:17:12: error: expected ';' after return statement
   return 0 // ; is missing!
           ^
           ;
2 warnings and 2 errors generated.
```

- Exact error message may vary depending on compiler version and setup


Watch the messages evolve as you fix one error at a time


What happens if you provide a string or a float for `nIterations`?

Checking input with `iostream`

Based on `examples/01/CinCheck.cpp`

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

int main() { // main begins here

   int nIterations = 0; // Initialized

   cout << "How many iterations? ";

   cin >> nIterations;

   // fails if input data does not match expected data type
   if(cin.fail()) cout << "cin failed!" << endl;

   // print message to STDOUT
   cout << "Number of requested iterations: " << nIterations << endl;

   return 0;

} // end of main
```

```
$ g++ -o cin_check cin_check.cpp
$ ./cin_check
How many iterations? 9
Number of requested iterations: 9
$ ./cin_check
How many iterations? asd
cin failed!
Number of requested iterations: 0
```

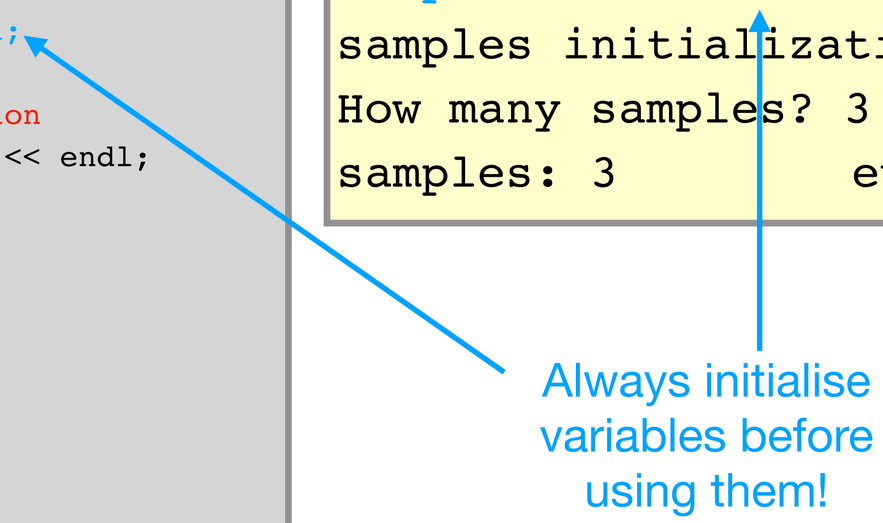
C Remarks Relevant for C++

Variable Declaration and Definition

Based on `examples/01/SimpleVars.cpp`

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

int main() {
    int samples; // declaration only
    int events = 1; // declaration and assignment

    cout << "samples default value: " << samples << endl;

    samples = 123; // assignment separate from declaration
    cout << "samples initialization value: " << samples << endl;

    cout << "How many samples? " ;
    cin >> samples; // assigment via I/O

    cout << "samples: " << samples
         << "\t"  // insert a tab in the printout
         <<  "events: " << events
         << endl;

    return 0;
} // end of main
```

```
$ g++ -o SimpleVars SimpleVars.cpp
$ ./SimpleVars
samples default value: 0
samples initialization value: 123
How many samples? 3
samples: 3        events: 1
```



Control Statements

Based on `examples/01/SimpleIf.cpp`

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

int main() { // main begins here

   if( 1 == 0 ) cout << "1==0" << endl;

   if( 7.2 >= 6.9 ) cout << "7.2 >= 6.9" << endl;

   // Declaring and initializing a boolean variable
   bool truth = (1 != 0);
   if(truth) cout << "1 != 0" << endl;

   if( ! ( 1.1 >= 1.2 ) ) cout << "1.1 < 1.2" << endl;

   return 0;
} // end of main
```

- What output do you expect from this code?

```
$ g++ -o SimpleIf SimpleIf.cpp
$ ./SimpleIf
7.2 >= 6.9
1 != 0
1.1 < 1.2
```

Loops

Based on `examples/01/SimpleLoop.cpp`

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

int main() { // main begins here

   int nIterations;

   cout << "How many iterations? ";
   cin >> nIterations;

   int step = 1;
   cout << "step of iteration? " ;
   cin >> step;

   for(int index = 0; index < nIterations; index += step) {
     cout << "index: " << index << endl;
   }

   return 0;
} // end of main
```

- Start
- Control condition
- Step

```
$ g++ -o SimpleLoop SimpleLoop.cpp
$ ./SimpleLoop
How many iterations? 7
step of iteration? 3
index: 0
index: 3
index: 6
```

Arrays

Based on `examples/01/SimpleArrays1.cpp`

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

int main() {

    float v1[3] = {0.4, 1.34, 56.156};
    float v2[3]; // use default value 0 for each element
    // array of size 7
    float v3[] = { 0.9, -0.1, -0.65, 1.012, 2.23, -0.67, 2.22 };

    for(int i = 0; i < 5; ++i) {
        cout << "i: " << i << "\t"
             << "v1[" << i << "]: " << v1[i] << "\t\t"
             << "v2[" << i << "]: " << v2[i] << "\t"
             << "v3[" << i << "]: " << v3[i]
             << endl;
   }

   return 0;
}
```

- Array indexing starts with 0
- Beware of accessing out of range array elements! The numbers in there will be whatever was stored last at that specific memory address

```
$ g++ -o SimpleArray1 SimpleArray1.cpp
$ ./SimpleArray1
i: 0  v1[0]: 0.4          v2[0]: 0     v3[0]: 0.9
i: 1  v1[1]: 1.34         v2[1]: 0     v3[1]: -0.1
i: 2  v1[2]: 56.156       v2[2]: 0     v3[2]: -0.65
i: 3  v1[3]: 0            v2[3]: 0.4   v3[3]: 1.012
i: 4  v1[4]: 4.62122e+35  v2[4]: 1.34  v3[4]: 2.23
```

Arrays and Pointers

Based on `examples/01/SimpleArrays2.cpp`

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

int main() {

    int v1[3] = { 1, 2, 3 };
    int v2[3]; // not initialized!
    int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

    int* c = v1;
    int* d = v3;
    int* e = v2;

    for(int i = 0; i < 5; ++i) {
        cout << "i: " << i << ", d = " << d << ", *d: " << *d;
        ++d;
        cout << ", e = " << e << ", *e: " << *e;
        ++e;
        cout << ", c = " << c << ", *c: " << *c << endl;
        ++c;
   }

   return 0;
}
```

- The name of the array is a pointer to the first element of the array
- Notice the addresses increasing by increments of 4 (in hexadecimal): this is the size of an `int` (more on this soon)
- We are accessing `v1` and `v2` out of range via `c` and `e`

```
$ g++ -o SimpleArray2 SimpleArray2.cpp
$ ./SimpleArray2
i: 0, d = 0x16ee0f700, *d: 1, e = 0x16ee0f71c, *e: 0, c = 0x16ee0f728, *c: 1
i: 1, d = 0x16ee0f704, *d: 2, e = 0x16ee0f720, *e: 0, c = 0x16ee0f72c, *c: 2
i: 2, d = 0x16ee0f708, *d: 3, e = 0x16ee0f724, *e: 0, c = 0x16ee0f730, *c: 3
i: 3, d = 0x16ee0f70c, *d: 4, e = 0x16ee0f728, *e: 1, c = 0x16ee0f734, *c: 0
i: 4, d = 0x16ee0f710, *d: 5, e = 0x16ee0f72c, *e: 2, c = 0x16ee0f738, *c: 905183414
```