A brief introduction to C++

James Richings, EPCC

j.richings@epcc.ed.ac.uk

Introduction

Assumptions

- You have some experience in another language
- You know how to use the shell
- You have access to a terminal in front of you with a C++ compiler

What this is not!

Writing efficient software, more than anything, requires you to choose an appropriate algorithmic approach for your problem.

Here we want to take a lower-level approach and talk about how to implement patterns efficiently using C++.

What is "scientific computing"?

Both HPC and data science, when you actually come to running a program, are about getting a large amount of data from memory to a core, doing something useful to it, and storing it again.

This is why FORTRAN is still relevant!

But it does force you to confront this all time.

C++ is all about building abstractions and composing them.

I will talk about a few and give some suggestions of default rules

These few lectures

We could spend a whole semester going in depth on C++, so we've picked a handful of features to cover that you really need to write modern C++ for technical computing.

This is not trying to teach the *whole language* in the many different styles that people have developed over the decades.

Please ask questions any time!

A bit of history

- C++ was originally designed as an extension of C
  - Well written C tends to be legal C++ but C is not a subset of C++
  - There are programs that are valid C but not valid C++
  - However, C++ supports every programming technique supported by C
- Released in 1985, but first standardised in 1998 with C++98
- New features added to the standard over time

The misunderstood monster

By Universal Studios - Dr. Macro, Public Domain, https://commons.wikimedia.org/w/index.php?curid=3558176

The misunderstood monster

- Large: the C++20 standard is just over 1800 pages
- Composed of many parts: C, classes, generics, functional programming, exceptions, the vast library, ...
- Inspires dread in those do not understand it
- Dangerous:

> "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off." - Bjarne Stroustrup

- "Expert friendly"

The misunderstood monster

- However!
- We can pick and choose the parts of the standard we want to use
- Not expected to know "all of C++"
- Pick a subset, write some code, and gradually learn more of the language, its libraries, and its tools

The philosophy of C++ (community)

- General purpose
- Flexible by allowing developers to build abstractions (and provides a large number through the library)
- Performance and efficiency are always targeted "you only pay for what you use"
- Use the powerful type system to express intent
- Communicate with the reader, not the compiler

C++ is alive!

- C++ is a work in progress.
- Every three years there is a new update to the International Standard (since C++11)
- C++20 is only fully supported by MSVC (although GCC is nearly there). Major new features are ranges, coroutines, concepts, and modules
- Latest one, C++23 has been released. Major features include networking, string formatting, executors, and consolidation of new C++20 features
- Good summary of compiler support here: https://en.cppreference.com/w/cpp/compiler_support

References

- Bjarne Stroustrup, "Programming: Principles and Practice Using C++" (2nd Ed.). Assumes very little but it's long
- Bjarne Stroustrup, "A Tour of C++". Assumes you're an experience programmer and is quite brief - targets C++17
- Best online *reference* is http://en.cppreference.com/ (comes in other human languages too!)
- cppreference.com also lists the level of C++ standards support (including C++20) for a number of compilers (e.g. GCC, Clang, MSVC, Intel C++)
- Scott Meyers, "Effective Modern C++", 2014. This is *the* book to get once you know your way around C++, but you want to improve. Teaches lots of techniques and rules of thumb for writing correct, idiomatic, maintainable code.
- stackoverflow.com has a lot of good questions about C++ (look for ones with at least 100 up-votes).

But first...

Hello!

```
#include <iostream>

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

Hello!

```cpp
#include <iostream>

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

```
$ g++ --std=c++17 hello.cpp -o hello
$ ./hello
Hello, world!
```

Hello!

```
#include <iostream>

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

- The preprocessor runs first
- The `#include` directive copies the contents of another file into the current compilation unit.
- The angle brackets `<...>` tell it to look only in system directories.
- This includes the `iostream` standard library header

Hello!

```
#include <iostream>

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

- Every program must have exactly one main function.
- The compiler and OS arrange for this to be called when you run it.
- The `return 0` statement indicates to the OS that no error occured.
- (You can also get the command line arguments but the empty brackets here means we aren't using them).

Hello!

```
#include <iostream>

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

- `std` is the standard library namespace.
- A namespace allows scoping of names (much like a filesystem has directories).
- The scope resolution operator `::` lets us access something from inside a namespace.
- `cout` represents console output (i.e. standard output / stdout)

Hello!

```
#include <iostream>

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

- The standard library uses the bitwise left shift operator (<<) to mean stream insertion:
  - i.e. output the right hand side to the left.
- Every statement in C++ must be terminated with a semicolon (;)
- The language treats all whitespace (space, tab, line break) the same

Your turn

Machine choice

You can use your laptop or ARCHER2

Your machine : you need a C++ compiler that supports at least C++11 (ideally C++17). If you use Windows and MSVC we may not be able to help much... Sorry!

ARCHER2: You have log in details.

Once connected you need to load the up-to-date compilers:

```
module load gcc/11.2.0
```

Getting the source code

All these slides and the exerices are available on GitHub:
https://github.com/EPCCed/archer2-cpp

You also can view the slides and other materials in a browser by going to
https://EPCCed.github.io/archer2-cpp

In the terminal, you need to use git get a copy of the exercises:

```
git clone https://github.com/EPCCed/archer2-cpp
```

Then you can change to the directory with this simple program

```
cd archer2-cpp/lectures/cpp-intro/hello
```

Say hello

View the program source

```
vim hello.cpp
emacs hello.cpp
```

Compile the program:

```
g++ --std=c++17 hello.cpp -o hello
```

No output means success!

Run it:

```
./hello
```

Compiler explorer (Godbolt)

A very useful tool is the Compiler Explorer - https://godbolt.org

Type some code on the left and it will compiled with your choice of very many compilers.

Will show compiler errors/warnings/reports and can execute your program.

Can get shareable links to your code.

A few things

C++:

- It is a typed language - all variables must be declared
  - But you can often tell the compiler to figure out the type
- Counts from zero (like C, Python) not from one (like Fortran)

Variables

Variables

A variable is

- an object

or

- a reference

that is declared to have a type and a name.

Objects

An object is a region of storage that has:

- type
- size
- value
- lifetime

Types

What is a type?

"Objects and expressions have a property called type, which both restricts the operations that are permitted for those entities and provides semantic meaning to the otherwise generic sequences of bits." -- https://en.cppreference.com

Fundamental types

Type	Description
`void`	Nothing - used to indicate a function returns no value.
`bool`	`true` or `false`
`int`	Standard *signed* integer for your machine. *At least* 16 bits. *Usually* 32 bits.
`double`	Double-precision floating point. *Usually* an IEEE 754 64 bit number.
`std::byte`	Raw untyped memory

There are also `unsigned` versions of the integer types

The header `<cstdint>` provides fixed-width integer types available on your implementation: e.g. `std::int32_t` and `std::uint64_t`.

Strings

The standard library has a class called `string` that holds a string of text characters.

You have to `#include <string>` to use it which includes the "header file" that contains all the information the compiler needs to let you use it.

```
#include <iostream>
#include <string>

int main() {
  std::string message = "Hello, world";
  std::cout << message << std::endl;
  return 0;
}
```

Conditionals

If

Use if to specify weather a block of code should be executed based on a condition.

```
if (condition) {
  //block of code
}
```

Conditions include:

- Equal to `a == b`
- Not equal to `a != b`
- Less than `a < b`
- Greater than `a > b`

Combining conditions

You can combine conditions together to make more complex logic:

- and

```
if (condition A && condition B) {
  //block of code
}
```

- or

```
if (condition A || condition B) {
  //block of code
}
```

if and else

The `else` statement allows you to specify what happens if the condition is false:

```
if (condition) {
  // block of code executes when condition true
}
else {
  // block of code executes when condition false
}
```

If we want more than else

More complex control flow can be generated with else if:

```
if (condition 1) {
  // block of code executes when condition 1 is true
}
else if (condition 2) {
  // block of code executes when condition 2 is true
}
else if (condition 3) {
  // block of code executes when condition 3 is true
}
else {
  // block of code executes when conditions are false
}
```

Shorthand if

In some cases you might want to be more compact

```
variable = (condition) ? expression if True : expression if False;
```

Switch statement

```
switch(expression)
  case a:
    // code block if expression = value a
    break;
  case b:
    // code block if expression = value b
    break;
    .
    .
    .
  default:
    // default code block if no cases matched
```

Functions

Functions

A function encapsulates a piece of code between braces (curly brackets, {}) and gives it a name so you can use it later.

```
void say_hello() {
  std::cout << "Hello, world!" << std::endl;
}

int main(int argc, char* argv[]) {
  say_hello();
  return 0;
}
```

Functions

You must declare the return type and parameter types.

Parameters are local variables that are initialised by the caller.

Return a value with the `return` statement - the type of the expression must be (convertible to) the declared return type.

```
int sum(int a, int b) {
  return a + b;
}
```

Since C++11, we can also use a *trailing return type* (aka "east end functions"):

```
auto sum(int a, int b) -> int {
  return a + b;
}
```

Functions

To use a function, or "call" it, you give its name and then provide the arguments in parentheses

```
int main () {
  int x = 42;
  std::cout << "Total = " << sum(x, 100) << std::endl;
  return 0;
}
```

The parameters to the function must match the declaration.

The `return 0` statement in the `main()` function is optional but you have to have a `return` statement in all other functions that return a value.

Function overloading

You can have multiple functions with the **same name** but **different arguments**.

```
int sum(int a, int b) {
  return a + b;
}

double sum(double a, double b) {
  return a + b;
}
```

When you call `sum`, the compiler knows the types of the arguments and will try to find the best match from all the candidates with the name.

The compiler will also try to use any built-in or user-defined conversion rules.

What happens here?

```
int i1 = 1;
int i2 = 2;
double d1 = 1.0;
double d2 = 2.0;
unsigned u42 = 42;
std::string name = "Alice";
std::string file = "data.csv";

std::cout << sum(i1, i2) << std::endl;
std::cout << sum(3, 72) << std::endl;
std::cout << sum(i1, u42) << std::endl;
std::cout << sum(d2, d1) << std::endl;
std::cout << sum(d2, 1e6) << std::endl;
std::cout << sum(d2, i1) << std::endl;
std::cout << sum(name, file) << std::endl;
```

Let's write some code

```
#include <iostream>
#include <string>

void say_hello(void) {
  std::cout << "Hello, world!" << std::endl;
}

int main(int argc, char* argv[]) {
  std::cout << "What is your name?" << std::endl;
  auto name = std::string{};
  // Read from the terminal
  std::cin >> name;

  // Have the program greet the user by name
  say_hello();

  return 0;
}
```

Exercise:

Change `say_hello` to accept the name it reads from the terminal, create a new message saying "Hello, $NAME!" and print it to standard output.