Programming in C++

Session 1 – Introduction

Dr Christos Kloukinas

City St George's, UoL
https://staff.city.ac.uk/c.kloukinas/cpp
(based on slides originally produced by Dr Ross Paterson)

What's this module about?

Goal Become a novice C++ programmer.

- That's actually advanced!
- Hard for novice programmers.

- C++ is hard
  - Multiple programming styles (procedural, OO, generic programming)
  - Language & compilers geared towards experienced programmers
    - Function calls are often hidden
    - Compiler messages can seem cryptic
  - Different standards: 1998, 2011 (major changes!), 2020, 2023
- Please ask questions!!! (lecture/Moodle)

This module: more OO programming, in C++

Assuming that you are a reasonably skillful Java/C#/etc. programmer, by the end of this course you should be able to

- read and modify substantial well-written C++ programs
- create classes and small programs in C++ that are:
  - Correct
  - Robust
  - Clear
  - Reusable
- use various object-oriented features, including genericity, inheritance and multiple inheritance

A bit of language history

1960 Algol 60: block structure, static typing
1967 Simula: Algol plus object-orientation (for simulation)
1970 C: statically typed procedural language with low-level features
1972 Smalltalk: object-orientation (for graphical interfaces), no static types
1985 C++: C + Object-Oriented features and (later) genericity
1995 Java: *"C++ greatly simplified"*

Procedural Algol 60, C, . . .
"To dress a young child you do X, Y, Z"

Object-Oriented Simula, Smalltalk, C++, Java, . . .
"To dress a grown up, you ask them to dress themselves"

A bit of language history — Part II

1972 C Procedural, static typing, low-level access

1985 C++ Your beloved (top) language C ***extended!***

- C++ compilers **can** compile C programs (The Linux kernel is compiled in this way)

C++ "C is good"

1995 Java Your beloved (top) language C++ ***simplified!***

- Java compilers **cannot** compile C++ programs

Java "C++ is ***too*** complex"

The differences between C++ & Java are serious pain points

One needs to understand them to understand the C++ language
(expert knowledge of Java not really required for this)

C++ design criteria

Started as "C with Classes"

- support a variety of programming styles, including object oriented (give the programmer more choices)
- powerful (give the programmer more control)
- enable efficient implementation (shift some implementation concerns to the programmer)
- extension of C (machine-level access)
  Often C features coexist with newer, cleaner versions.
  And C++98 features coexist with C++11 & C++20 versions. . .

Java design criteria

Keep things as simple as possible

- object orientation
- (moderate) simplicity (fewer variant ways of doing things)
- robustness and security (type-safe, automatic memory allocation)
- architecture-neutral (fairly high level)
- syntax based on C++

This session: non-OO programming in C++

This session introduces the philosophy of C++, and some simple non-OO programs.
We will touch on the following features of C++:

- Operator overloading
- Constants
- Initialization vs. assignment ⋆⋆
- Parameter passing by value and reference ⋆⋆
- Some library classes

All will be explored in greater detail later.

⋆⋆ NOT like Java!

The toolset

To	Java	C++
Compile (notes)	`javac -g pkg1/pkg2/.../pkgN/X.java` `-g` debug on	`g++ -g -c x.cpp` `-c` compile only

Link/etc
 or
 (notes)
	`jar cfe prog.jar X X.class`
	 `echo Main-Class: X > manifest.txt`
	 `jar cfm prog.jar manifest.txt X.class`
	 e executable (“`main`” is in class `X`)
		`g++ -g -o prog x.o`
		 `-o` output to

Execute	`java -jar prog.jar`	`./prog`

Debug	`jdb -classpath prog.jar X`
	 `stop in X.main`
	 `run a1 a2 a3`
	 `print 3+4`
	 `print args`
	 `step`
		`gdb prog`
		 `break main`
		 `run a1 a2 a3`
		 `print 3+4`
		 `print argv[0]`
		 `step`

Curious	`javap -c X`	`nm -C x.o` `nm x.o | c++filt`

A C++ program is processed by the preprocessor (**cpp**), the compiler (**g++**), and the linker (**ld**) – all of these can complain.

A small C++ program (vs in Java)

```
        /* C++: */
#include <iostream>
using namespace std;

int main(int argc
         , char *argv[]) {
 cout << "Hello world!\n";
 return 0;
}
```

```
        /* Java: */

class MyProg {
 public static void main(
  String[] args){
   System.out
    .print("Hello world!\n");
 }
}
```

- The first two lines make available names from the standard library, like **cout**. *C++ pretends to not know the standard types...*
- In C++ (like C), a function (**main**) can exist outside of any class.
  - Java: oh, that's a (**public**) **static** method! *Q: Called on what?*
- Style: C++ – **lower_case**, Java – **CamelCase**

*Q: Where's the **print** function call in C++?*

*Q: Java: Where do **args** come from?*

Accessing names from standard libraries

- In Java, classes are collected in packages, and accessed with **import** declarations.
- In C++, there are two (mostly) independent ways of controlling access to names:

  header files — like **iostream** contain collections of related definitions (in this case for I/O streams). A typical program will begin with several **#include** lines.

  namespaces — like **std** are collections of names, which must usually be qualified (**std::cout**), unless there is a **using** command.
  Each source file will include the above **using** line, but we will not make any other use of namespaces.

Text output

```
cout << "Hello world!\n";
```

- The **iostream** header defines three standard streams:
  - cin — standard input (cf. Java's **System.in**)
  - cout — standard output (cf. Java's **System.out**)
  - cerr — error output (cf. Java's **System.err**)
- Applied to integers, **<<** performs a left shift (as in Java)
- Applied to an output stream and a string, writes the string to the stream
- The **<<** operator is *overloaded*

- Why do we need both **cout** and **cerr**?
  - We need both so that we can separate the output from the errors into different files (or sockets), *e.g.*, when using the bash command shell:
    program > output.txt 2> errors.txt
- What's the difference between **cout** and **cerr**? Why would one want to use both if not splitting the output as above?
  - We need both because they behave differently.
  - When printing to **cout**, our output is *buffered*, *i.e.*, it is placed into a temporary area and stays there until the output buffer has been filled. When the buffer is full, the output is sent out to wherever it is supposed to be sent (terminal, file, network).
  - Unlike **cout**, when printing to **cerr** the output is not buffered – it is printed immediately.
  - This is why when printing to **cout** we sometimes have to use **flush** to tell the buffer to output whatever it has stored, even if it is not full:
    cout << "Hi"; cout.flush();
    Or alternatively:
    cout << "Hi" << flush;

Flushing streams – endl

- Another way to flush the output stream is to use **endl**. We've seen so far how to use the special character **'\n'** to insert a newline character into the output. With **endl** we can insert a newline and at the same time flush the output stream:

```
cout << "Hello, how are you?\n" // no printing yet
     << "How could I be of assistance?"
     << endl; // Add a new line & flush everything
```


Input and output

```
int i;
cout << "Type a number: " << flush;
cin >> i;
cout << i << " times 3 is " << (i*3) << '\n';
```

- The >> operator reads from an input stream.
- The << operator associates to the left, and returns the stream; the above is equivalent to
  (((cout << i) << " times 3 is ") << (i*3)) << '\n';
- It is also overloaded for **int** (**i**, **i*3**) and **char** (**'\n'**).
- The >> operator is similar.


```
  cout << i  << " times 3 is "  << (i*3)  << '\n';// same
(((cout << i) << " times 3 is ") << (i*3)) << '\n';
```

In order for this to work, the **operator<<** has to return an output stream. That's why when **(cout << i)** is computed we can use its result (the modified **cout** (**cout'**) to apply the next **operator<<** with the next argument (**" times 3 is "**).
So:

```
(((cout << i) << " times 3 is ") << (i*3)) << '\n';
       cout'  << " times 3 is "
             cout''               << (i*3)
                                 cout'''    << '\n';
```

Strings

```
#include <string>
```

The standard library provides a **`string`** type:

```
string s = "fred";
cout << s;
cin >> s;     // reads a word
```

The **+** operator is overloaded on strings:

```
s = s + " and bill";
s = s + ',';
```

So are **+=**, **==**, **<**, etc.

Unlike in Java, strings are modifiable:

```
s.erase();    // now s == ""
```


Breaking the input into words

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

int main() {
        string s;
        while (cin >> s)
                cout << s << '\n';
        return 0;
}
```

- The >> operator on strings reads words.
- ***The stream returned by the >> operator can be used in a conditional, to test if the read was successful.***

(what do these words mean?)

Breaking the input into words

`while (cin >> s)`
"***The stream returned by the >> operator can be used in a conditional, to test if the read was successful.***" *?!?!*
The expression **`cin >> s`** returns the modified input stream **`cin`**', which is what we ask **`while`** to evaluate so as to decide whether the loop body should be executed or not.
The C++ library has functions that allow one to translate an input stream into a boolean – the boolean is true if the last attempt to read from the stream succeeded, and it's false otherwise (*e.g.*, the input had finished, the input is corrupted, *etc.*). These functions work like when we write **`s1 = s2 + " Hi " + 3;`** in Java – there they translate automatically the array of characters **`" Hi "`** and the integer **`3`** into string objects, that they concatenate with the string object ***referenced*** by **`s2`** to obtain the value of the string object that will be ***referenced*** by **`s1`** *(**`s1`** and **`s2`** are not objects in Java, they are **pointing** to objects.)*.
The meaning of **`while (cin >> s)`** is:
"Try to read a word from **`cin`** into string object **`s`** and if that has succeeded, then continue executing the body of the while loop."

Vectors

```
#include <vector>
```

C++ has arrays, but we'll use vectors instead (*cf.* Java's **`ArrayList`**):

```
vector<int> vi(5);   // vector of 5 ints
vector<string> si;   // empty vector of strings
```

Vectors can be accessed just like arrays:

```
vi[1] = x;         // vi.set(1, x);   <3 Java! :-P
vi[2] = vi[1] + 3;// vi.set(2, vi.get(1) + 3); <3 <3
```

Vectors can also be extended:

```
si.push_back(s);
```

The current length of **`si`** is **`si.size()`**

Syntax seems simple but the meaning is not. . .

Expression “`vi[1]`” in Java would have to be written as “`vi.get(1)`”, where `vi` would have been declared instead as a Java pointer to an `ArrayList` container.

- Thanks to operator overloading C++ allows us to type less (2 characters for “`[]`” instead of 6 characters for “`.get()`”.
- It also allows us to keep the syntax of arrays that we're familiar with and treat vectors as if they're advanced arrays (that we can extend/shorten).
- But this comes at a price – the code is not as clear now as it was in Java. In Java it's obvious we're calling a function while in C++ it is not so obvious – one has to remember that ***every*** use of an operator is actually a function call in C++!
- So `vi[1]` is actually `vi.operator[](1)`.

Language notes

- `string` is a class
- `vector` is a template (generic) class
- C++ has pointers (like in Java), but we won't use them till later:

```
string s1 = "bill", s2;
```

declares (and initializes) string objects, not pointers

- assignments like

```
s1 = s2;
```

copy the objects (not the Java pointers!)

Note: syntax looks like Java, but meaning is VERY different

Capitalisation: In C++ everything is lower case – words are separated by underscores: `class string`, `void push_back`

Initialization vs. assignment

Initialization of variables:

```
string s1;
string s2 = "bill";
```

Objects are always initialized; variables of primitive type aren't.
Assignment replaces an existing value:

```
s1 = s2;
```

Initialization defines a new variable:

```
string s3 = s2;
```

Slide has 4 different method calls!
(C++ function calls are often hidden!)

SUPER IMPORTANT!!! – I

This slides looks simple and boring – initialise some variables, assign some variables, blah blah blah, whatever. . .
Your success in the module depends on understanding it fully – and it ain't easy.
It actually shows four different methods.
Remember that `s1`, `s2`, and `s3` are real objects in C++ – unlike Java where they are *pointers*.

```
string s1;
/* INITIALISATION: To initialise s1, the string
constructor must be called.
Which constructor? The one taking no arguments.
So here, we call:
string()
           SPECIAL NAME: ``Default Constructor'' */
```

SUPER IMPORTANT!!! – II

```
string s2 = "bill";
/* INITIALISATION: Which constructor do we call to
initialise s2?
The one taking an array of characters:
string( const char a[] ) */

s1 = s2;
/* ASSIGNMENT: s1 and s2 are OBJECTS, not just
pointers to objects (as in Java).

So here we're calling a FUNCTION:
string & operator=(string &o, const string &o);
Though usually we're calling a METHOD:
string & operator=(const string &o);
          SPECIAL NAME: ``Assignment Operator'' */

string s3 = s2;
/* INITIALISATION: Which constructor do we call
to initialise s3?
The one taking another object of class string:
string( const string &o )
             SPECIAL NAME: ``Copy Constructor'' */
```


Is it initialisation or assignment?

- To distinguish between initialisation and assignment you need to look at the form of the statement.
- If it's initialisation we are just introducing a new variable, so we have to tell the compiler what is its type.
  ```
  string s1;
  string s2 = "Bill";
  string s3 = s2;
  ```
  All initialisations of objects call a constructor of the object's class.
- When assigning a variable the variable exists already, so we do not declare its type:
  ```
  s1 = s2;
  ```
  Assignments call the assignment operator: `operator=`

The BIG Difference

Java	C++
`String s;`	`string s;`
`// s == null`	`// s != null`
`// s is a Java *POINTER*!!!`	`// s is an OBJECT`
`// nothing called`	`// constructor called!`

- You can never access an object directly in Java (for *safety*).
- C++ gives you direct access to objects (for *performance/control*).

Many of their core differences are a consequence of this!

- Garbage collection ***vs*** Manual memory deallocation
- Sharing objects by copying Java pointers ***vs*** Copying objects
- Immutable strings ***vs*** Modifiable strings
- Call by value ***vs*** Call by reference

DANGER!!!

If you don't understand what the big difference is here, you're in dangerous waters.

- Draw a picture of the memory for Java and another for C++.
- Draw the objects in each – there is one for Java and one for C++.
- The C++ object is called `s` – that's all there is in the memory of C++.
- The Java object has NO NAME. In Java, the name `s` is the name of an object POINTER [*], and this (Java) POINTER is in another location in memory and is pointing to the actual Java object.

Confused? Go over this again (and again, and again, . . .) till you have understood it – it's super-basic and you'll suffer if you don't get it.

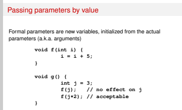
[*] Java's "references" are ***pointers*** – that's why when you try to use a NULL Java "reference" you get a "`NullPointerException`". You do not get a "`NullReferenceException`", do you?

Passing parameters by value

Formal parameters are new variables, initialized from the actual parameters (a.k.a. arguments)

```
void f(int i) {
        i = i + 5;
}

void g() {
        int j = 3;
        f(j);   // no effect on j
        f(j*2); // acceptable
}
```

Pass by value

- `void f(int i)` – here `i` is a ***local*** variable of function `f`, which gets initialised with whatever we pass as argument to the function.
- That's why we can call the function with an expression as an argument:
  `f( 3 * 2 );`
  Parameter `i` will be initialised with the value of that expression
  `int i = 3*2; /* 6 */`

Parameter passing in Java

- In Java/C, all parameters are passed by value
  Even Java/C pointers!

By Value: the method is given ***a copy*** of the parameter

  - Any changes have no effect on the original

- If param is a pointer, the copy *points* to the same object
  - So it is possible to modify the object *pointed* to

But: Cannot modify original pointer to point to another object

Limitations of value parameters

- Might wish to change an actual parameter inside a function
- Parameter might be large (e.g. an object), so expensive to copy
- A solution (Fortran, Pascal, C++, etc) is reference parameters
- Can get similar effects with value parameters & pointers
  (but more error-prone)
  (so we prefer references)

Passing parameters by reference

A *reference* parameter is another name (an alias) for the actual parameter

```
void f(int &i) {
        i = i + 5;
}

void g() {
        int j = 3;
        f(j);   // j is updated
        // f(j*2);   // NOT ALLOWED!
}
```

Note: There is no relationship to Java's pointers ("references").

Less error prone: Reference params can never be NULL!

Passing large values by reference

Reference parameters are also used to avoid copying large values:

```
int last(vector<int> &v) { // v exists!
        return v[v.size() - 1];
}

void g() {
        vector<int> x(100);
        ...
        int n = last(x);   // don't copy x
}
```


Constant parameters: `const` <3 <3 <3

We can indicate that the function doesn't change the parameter with the keyword **`const`**:

```
int last(const vector<int> &v) {
        return v[v.size() - 1];
}

void g() {
        vector<int> x(100);
        ...
        int n = last(x);   // don't copy x
}
```

This makes programs **safer**, and **helps** the compiler.

Constants

- The C++ keyword **`const`** introduces a *constant*:

  ```
  const int days_per_week = 7;
  ```
- Constants may (must!) be initialized, but cannot be assigned to.
- **`const`** parameters are a special case.
- C programmers: use **`const`** instead of **`#define`**, or use **`enum`** definitions:

  ```
  enum class traffic_light { red, yellow, green };
  traffic_light r = traffic_light::red;

  enum class colour_rgb { red, green, blue };
  colour_rgb r = colour_rgb::red;
  ```
- A different use of **`const`** will be mentioned later.

Use `const` wherever you can!

We should always try to use **`const`** wherever we can and only remove it if the compiler complains that we cannot update something because it is const (and we cannot figure another way to do what we want without updating).
Consts improve our code — make it more robust and help the compiler optimise further.
Other ways to restrict the code and help the compiler is to use the more restrictive versions of things, e.g., (lecture 7) prefer **`unique_ptr<T>`** over **`shared_ptr<T>`**, if possible.

John Carmack (founder and technical director of Id Software) had written a blog post (back in 2013) about this — read it here: https://web.archive.org/web/20130819160454/http://www.altdevblogaday.com/2012/04/26/functional-programming-in-c/
In his Quakecon 2013 keynote he also talked about it (among other things) — this is the relevant part: https://www.youtube.com/watch?v=1PhArSujR_A

References

- The C++ symbol **`&`** after a type defines a ***reference***, which is another name (or alias) for a piece of ***storage (a.k.a. lhs)***
- Initialization defines the reference as an alias:

```
int x;
int &y = x; // there's only one int here

person dr_jekyll;
person & mr_hyde = dr_jekyll; // only one person
```

- Assignment assigns to the original storage:

```
y = 3;
```

  is the same as assigning to **`x`**.

References can never be NULL!

- C++ references are *almost* like (const) pointers:
  - A reference can never be **`NULL`** - it must always refer to a legitimate object;
  - Once established, a reference can never be changed so that it refers to a different object - a **`const`** pointer;
  - A reference does not require any explicit mechanism to de-reference the memory address & access data values (it's just an alias).
- C++ references are NOT pointers.
  - Never state in public or write down that they are pointers.
  - Never say that they "point" to an object or say that they "have its address".
    All of these demonstrate a gross misunderstanding of what a C++ reference is.
    A C++ reference IS the thing it refers to. They are one and the same.
- Why use references inside a block of code? To simplify things:

```
int &size = tree.left.value.size;
++size;
cout << size;
```

  equivalent to:

```
++(tree.left.value.size);
cout << tree.left.value.size;
```


An example function (from `iostream`)

```
istream & getline(istream & in, string & s) {
        s.erase();
        char c;
        while (in.get(c) && c != '\n')
                s += c;
        return in;
}
// Use:
//string s;while ( getline(cin, s) ){cout<<s<<endl;}
```

Note that

- **`get`** also uses pass-by-reference
- There's no copying here: arg **`in`** returned by reference
  (Cannot return a local by reference)

(never use `getline` unless explicitly told to)

An example function (from **iostream**)

```
istream & getline(istream & in, string & s) {
        s.erase();
        char c;
        while (in.get(c) && c != '\n')
                s += c;
        return in;
}
// Use:
//string s;while ( getline(cin, s) ){cout<<s<<endl;}
```

Note that

- **get** also uses pass-by-reference
- There's no copying here: arg **in** returned by reference
  (Cannot return a local by reference)

(never use **getline** *unless explicitly told to)*

- How many things does **getline** return? Three – the result, the modified parameter **in** and the modified parameter **s**.

  By using reference parameters you can return multiple things.
- Parameter **in** is passed by reference, because we need to modify the input stream (we modify it when we call **in.get(c)** since we remove one character from it).
- Parameter **s** is passed by reference because we need again to modify the string so as to be able to return to our caller the contents of the line we've read from the input.
- We cannot simply return a **string** from the function, because we need to return a stream – and we need that because we want to use **getline** as in the next slide, where we test the returned stream to see if **getline** succeeded in reading a line or note.
- Note that the returned result (**istream &**) is also returned by reference to avoid returning a copy of **in**!
- In order to return a variable by reference, the variable must not be local – it must have been received as a reference parameter.
  - This is because all local variables are destroyed when a function returns so they no longer exist to be returned themselves – only a copy of them can be returned.

2024-11-13
Programming in C++
└Examples
└An example function (from **iostream**)

An example function (from **iostream**)

```
istream & getline(istream & in, string & s) {
        s.erase();
        char c;
        while (in.get(c) && c != '\n')
                s += c;
        return in;
}
// Use:
//string s;while ( getline(cin, s) ){cout<<s<<endl;}
```

Note that

- **get** also uses pass-by-reference
- There's no copying here: arg **in** returned by reference
  (Cannot return a local by reference)

(never use **getline** *unless explicitly told to)*

(Advanced)

Since C++11, one can return an object without copying it. These versions of the C++ language standard support ***moving*** objects.

- If your class contains sub-objects of classes that are well-behaved (**string**, **vector<T>**, *etc.*) then objects of your class can be moved without you having to do anything special.
- Just pass flag **-std=c++23** to the compiler (this flag works for the **g++** and **clang++** compilers).

Prefixing lines with their lengths

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

using namespace std;

int main() {
        string s;
        while (getline(cin, s))
              cout << s.size() << '\t' << s << '\n';
        return 0;
}
```


Next session

- C++ Classes: very similar to Java, but with important differences.
- Reading:
  - *Absolute C++* by Walter Savitch, Addison-Wesley Longman, Reading, Mass, 2002. Chapter 1, sections 6.2 and 7.1.
  - *The C++ Programming Language* (3rd edition) by Bjarne Stroustrup, Addison-Wesley Longman.
    - For this session: sections 2.1–3 (except 2.3.3), 3.2–6 (except 3.5.1), 3.7.1.
    - For next session: sections 2.5.3–4, 2.6, 10.2.1–6.

Final Notes

- Make sure you understand the difference between initialisation **`(TYPE VARNAME = EXPRESSION;)`** and assignment **`(VARNAME = EXPRESSION;)`**.
  In C++ these call different methods – you need to know which case it is to figure out which method will be called (and to understand how to write these methods – more later).
- **BIG DIFFERENCE** between Java and C++ – in C++ you have direct access to objects, in Java you can only access **pointers** to objects.
- Because of the direct access to objects, C++ supports **call-by-reference** as well as **call-by-value** – make sure you understand the differences!
  (and call-by-constant-reference. . .)
  (and return-by-reference vs return-by-value. . .)