C++ London University Session 25

Tristan Brindle

Feedback and Communication

- Your feedback is vital
- Otherwise, we don’t know what you don’t know!

- Please join the #ug_uk_cpplondonuni channel on the cpplang Slack — Go to https://cpplang.now.sh/ for an “invitation”

Today's Lesson Plan

- Revision!
    - Feel free to ask questions!
- Nought-and-crosses exercise (time permitting)

Initialisation

- Fundamental types like `int` are not initialised by default
- Attempting to read an uninitialised variable is undefined behaviour
- The compiler will not give you an error message (except in a `constexpr` context), but if you're lucky it may warn you
- **Always initialise your variables!**

Initialisation (2)

- (Non-aggregate) class types like `std::string` will call their default constructor when constructed with no arguments
- However, it's not always obvious what's a class and what isn't
- **Always initialise your variables!**

Almost Always Auto (?)

- C++11 introduced `auto` for declaring variables with type deduction

- With `auto`, it is impossible to forget to initialise a variable

- Some authors recommend using auto pervasively for this reason — so called “Almost Always Auto” style

- (The contrary opinion: `auto` hides too much information and makes code unclear. Use it only when necessary, or when the type is obvious from the context.)

Special Member Functions

- The default constructor, destructor, copy and move constructors and copy and move assignment operators are collectively called the *special member functions*

- If you don’t declare a special member function, the compiler will generate it for you

- Declaring move operations will prevent copy operations from being generated, and vice versa

- You can request the compiler to generate a special member for you by using the `= default` syntax

Special Member Functions

- Remember the *Rule of Five*: if you use a non-default version of any of the special members (other than the default constructor), you probably need all five

- Much better is the *Rule of Zero*: let the compiler generate all the special members for you

- Smart pointers like `std::unique_ptr` help with this

Operator Overloading

- Almost all operators in C++ may be overloaded
- The most common are `operator=` (for assignment) and `operator==` and `!=` (for comparison)
- If your class has value semantics, it's usually a good idea to implement `operator==` and `!=`
- Golden rule: **don't surprise your users!**
- If you choose to overload an operator, make sure it provides the same semantics as for built-in types

Inheritance

- Inheritance allows you to separate the interface of a class from its implementation

- `structs` default to public inheritance, `classes` to private inheritance — public is almost always what you want

- In large projects, using separate interface and implementation classes is often a good idea

Multiple Inheritance

- Unlike many OO languages, C++ allows multiple inheritance
- This can be useful, but can lead to the “dreaded diamond” problem
- Virtual inheritance is a way out, but can be a sign that your class hierarchy is too complicated
- Multiple inheritance from pure interface classes (with no data members) is usually okay

Virtual Functions

- A “pure virtual” (“abstract”) method has the signature followed by the strange `= 0` marker

- A class with a pure virtual method cannot be instantiated directly

- If you are overriding a virtual method, use the “keyword” `override` — this will cause a compile error if you get the signature wrong

- Always provide a virtual destructor (`= default`) if your class has at least one other virtual method

Final Classes and Methods

- It's possible to mark a class as `final`, in which case it is not possible to inherit from it
- This can be an optimisation in some circumstances (it helps the compiler with devirtualisation)
- It can be a pessimisation in other circumstances
- It's probably best to avoid in general
- It's also possible to mark individual methods as `final`, which will prevent derived classes from overriding them

Pointers

- A pointer is a value representing a memory address of some other variable

- Pointers themselves have *value semantics* — they can be copied, assigned to, compared to see if they represent the same address, etc etc

- Pointers can also be dereferenced (via `*ptr` and `ptr->member`) to access the variable that they point to

- `nullptr` is a special variable, convertible to all other pointer types, which represents an “invalid address”

- `void*` is an untyped pointer: common in C code, but C++ gives safer alternatives

When to use raw pointers

- Raw (“non-smart”) pointers are still useful in modern C++!
- …but never to represent an “owned” resource
- Examples of good uses of raw pointers
  - As an “optional reference” in function interfaces
  - As an iterator for a C array
  - As a “rebindable reference”
- General rule of thumb: *use references when you can, pointers when you have to*

Smart Pointers

- Smart pointers are class templates that have “pointer-like” semantics, but typically manage ownership as well

- The standard library has two smart pointers, `std::unique_ptr` and `std::shared_ptr`

- `std::unique_ptr` should be your default, go-to type whenever you need dynamic allocation

- Use `std::make_unique()` in C++14 to create `unique_ptr`s

- `unique_ptr` is a move-only type: it cannot be copied. Use a virtual `clone()` method to copy a polymorphic type.

Smart Pointers

- `shared_ptr` represents shared ownership of a resource
- `shared_ptrs` are reference counted:
  - Copying a shared_ptr increments the reference count
  - Destroying a shared_ptr decrements the reference count
  - When the reference count reaches zero, the resource is released
- Shared pointers can act like a simple form of garbage collection…
- …that performs very poorly compared to “real” garbage collectors or built-in refcounting support
- Overuse of shared_ptrs can lead to reference cycles — use `std::weak_ptr` to break these
- Guideline: prefer `std::unique_ptr` by default. Use `shared_ptr` only when shared ownership is genuinely required, for example when passing data between threads

Templates

- Templates allow us to write code that is generic, type-safe and highly efficient

- C++ templates look similar to generics in languages like Java and C#, but the way they work is quite different

- A template is like a “blueprint” for how to write a class or function

- To use a template, we need to instantiate it with particular template arguments

Should I use templates?

- If you find yourself writing the same code (or almost the same code) several times for different types, it can be worth making it a template

- Upside: code re-use with type safety and unbeatable efficiency

- Downside: public templates must be defined in headers

- Downside: template error messages can inscrutable

- Downside: increased compile times

- Downside: potential for “code bloat” and increased executable size

Standard Containers

- The standard library provides a variety of containers. Get to know the tradeoffs each container offers
- For example, `std::vector` has fast iteration and append, but insertion elsewhere is (algorithmically) slow; `std::list` has slow iteration, but insertion anywhere is O(1)
- Use `std::map` or `std::unordered_map` for associative arrays

Choosing a container

- Guideline: use `std::vector` whenever you can
- Guideline 2: use `std::vector`
- Guideline 3: no, really, use `std::vector`
- Exceptions: use `std::string` for character strings, `std::array` where the number of elements is small and fixed at compile-time

Standard Algorithms

- The standard library provides a wide selection of generic algorithms

- Get to know the algorithms that are available

- Prefer using a standard algorithm to writing your own implementation

- Rule of thumb: *if you find yourself writing a non-trivial loop, think about whether you can use an algorithm instead*

Move Semantics

- In C++11, we can use move semantics to “steal” the contents of an object

- This is mostly useful for types which use dynamic allocation, like `vector`, `string` and `unique_ptr`

- For trivial types (e.g. `int`), a move is the same as a copy

Move Semantics

- Use `std::move()` to mark a variable as “available for moving” (technically: turn an *lvalue* into an *xvalue*)

- Note that `std::move()` doesn’t actually move anything by itself!

- In general, don’t `return std::move(x)`, where x is a local variable — the compiler will already avoid a copy/move if possible

- In general, once an object has been moved-from, it cannot be used again: it must be re-initialised or destroyed

rvalue References

- rvalue references are the machinery that powers move semantics
- However, you rarely need to use them directly except in move constructors and move-assignment operators
- If you are writing a “sink” function which stores the object passed to it, you can usually take the function parameter by value
- It’s almost never necessary to declare a local variable of rvalue reference type: use a non-reference object instead

Questions

- Ask us anything!

Exercise

- Back in week 4 we presented a group exercise which involved writing a noughts and crosses (tic tac toe) game
- With lots of new faces and lots more experience under our belts, it’s time to try again!
- This may or may not be useful for the assessment 😉
- https://github.com/CPPLondonUni/noughts_and_crosses

Online Resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)

Thanks for coming!

C++ London University:

- Website: cpplondonuni.com
- Twitter: @cpplondonuni
- Github: github.com/CPPLondonUni
- Reddit: reddit.com/r/CppLondonUni

Where to find Tom Breza:

- On Slack: cpplang.slack.com #learn #ug_uk_cpplondonuni
- E-mail: tom@PCServiceGroup.co.uk
- Mobile: 07947451167

My stuff:

- Website: tristanbrindle.com
- Twitter: @tristanbrindle
- Github: github.com/tcbrindle

See you next time! 🙂