C++

Basics

- `for(auto v: vArray)` - range over an iterable without explicit index
- `enum class` - Doesn't explicitly map to int, safer and recommended
- `func(&int)` - Reference to int, like pointer but dereferencing happens automatically and introduces some restrictions similar to how the `const` keyword restricts modification, recommended instead of pointers
- Always pair unions with an enum to represent which type its supposed to take on
- Properties of an object accessed with `object.property`, of a pointer to an object using `pointer->property` (or simply `.` if it is a reference `&` not raw pointer), and members of a class are accessed using `std::cout`
- `lvalue` - an object that occupies some identifiable location in memory and can be assigned to
- `rvalue` - expressions that aren't `lvalue`s, an object that isn't in memory, usually on the RHS in expression
- Expression evaluation may be short circuited (compiler optimization). For example: `1 != 2 && 6/3 == 1` will skip the right hand side division, unless it contains an assignment or something that could affect the control flow.
- The iteration expression is executed after the condition and the loop body have been executed, unlike JS
- The Turing Halting problem is the reason why not all programs (even those without external inputs) cannot be checked for errors / termination without executing the entire program (hence run-time errors exist)
- `switch - case` statements fall through unless break is used.
- Value types such as int, double, bool are passed by values to functions - new copies of them are created on the stack and the original values cannot be accessed directly from within the function scope
- Pre / postconditions specify the valid domain / expected range of a function (for example when dividing by an argument, it must not equal 0). These can be simply written as a comment `// PRE: condition` or expressed using `assert (e>=0 || b!=0)` by importing `#include <cassert>`, a sufficiently significant and easy to express pre / post condition is worth while expressing programatically.
- Functions are only visible after they have been declared, which can be problematic if two functions call each other. A solution is to firstly declare a function `int g();` defining its return and argument types, from which point it can be used (it's in scope), despite the implementation `int g() {}` being somewhere later in the file.

Lists

- Vectors are resizable `array` wrappers and usually used since they're far more flexible. Similar to `List<Type>` in `C#`
- `vector.at(index)` allows safe index access, throwing an error if the given index is out of bounds. A compiler flag can be enabled so regular indexing has the same effect instead of simply reading the memory location.
- Most standard library functions treat indexes as uints, which can lead to underflow problems when used in operations with regular ints
- Long type names can be aliased: `using matrix = std::vector<std::vector<int>>;`, although `auto` may be more appropriate in one-off cases.

- Vectors should be passed by reference `func(std::vector<int>& numbers)` to avoid copying lots of data to the stack. Arrays are automatically passed by reference, but I'm not sure about their exact functionality yet
- Const references are especially useful for vectors, preventing the copying of the data but avoiding modification of the original array
- In modern C++, a for each loop can be used `for(int number: numbers)`

Modularity

- Rather than including source `.cpp` files, using header files ensures that the "library" is only compiled once. Alternatively it allows calling functions from pre-compiled .obj files (for example from a closed-source library)
- All source files referenced are compiled separately, with non-main files being compiled into an `.obj` file with their implementations. A linker then copies implementations into the missing usages to create the final binary.
- Use the same header file for implementation and usage
- Errors are thrown to allow the user of a library to decide how to handle unexpected cases
- Namespaces group functions to prevent duplicate naming issues. They are referenced as follows: `namespace_name::func_name()`
- An entire namespace can be imported (functions available without prefix) through `using namespace` but this obscures the origin of a function and is recommend against.

OOP

- Use `struct` for plain data structures without any access modifiers or OOP features, otherwise use richer objects of a `class`
- Struct assignment `myType item = a;` copies the members of `a`, unlike JS
- `complex operator+(complex a, complex b) { return a+=b; }` overloading default operators
- `new` - Assigns memory on the heap for the object and returns a pointer. Has to be explicitly deleted (even after it leaves scope). Useful to allow a variable to be accessed by its pointer from outside of the current scope (otherwise it'll be automatically deleted).
- Concrete classes - Same as built in types, constructor initializes any needed heap properties and `~Destructor()` is called if `delete` is called to deallocate (unreserve) it .
- representation - the properties / variables of a class, what stores memory
- abstract class, similar to an interface in Go, simply a collection of methods such a class must implement, can be used to specify what an argument is expected to have. Implemented as `class Implementor: public Abstract {}`, this is **inheritance**
- Polymorphism - one interface used to represent many other types which may satisfy it
- `virtual` - May be redefined later in a derived class, `virtual void x = 0` means it **must** be redefined otherwise the class cannot be instantiated, there is no default implementation.
- Base functions / properties can be accessed within subclass implementations
- Calling `delete` on an abstract object calls the destructor of the shallowest subclass (as it has access to the most "extra" properties)
- `dynamic_cast` can be used to check what derived class an abstract argument is
- Resource handle - A class that is responsible for managing underlying resources, these provide custom copy implementations to prevent violating validity, for example assigning a `vector` to another variable results in two vectors that refer to the **same** underlying elements. Such handles should implement a **copy constructor** and **copy assignment** operator `Vector& operator=(const Vector& a)` so underlying resources are correctly reallocated.
- Marking a constructor `explicit` prevents automatic type conversion

- Default copy / move operations in a parent class can be deleted using `Shape& operator=(const Shape&) =delete;`

Generics

- Prefixing a class / function with `template<typename T>` accepts a type as a generic argument, so that `T` can be used throughout implementations
- So-called function objects can be defined by implementing the () operator, for example `bool operator()(const T& x) const { return x<val; }`
- Type aliases can be defined: `using value_type = T` is a public property of all container classes in the standard library, accessed using `Class::value_type`

Floating-point numbers

Floating point number systems are how types such as `double` and `float` represent real number approximations. Such systems allow storing and working with numbers in vastly different orders of magnitude and are denoted as follows:

$$F^*(\beta, p, e_{\min}, e_{\max})$$
$$\pm \sum_{i=0}^{p-1} (d_i \beta^{-i}) \times \beta^e$$
$$d_0.d_1, ..., d_{p-1} \times \beta^e$$

Where the digits are called the *mantissa* and the exponent e indicates the order of magnitude, as in scientific notation 1.6×10^{-19}:

- β - base (for example 10 - decimal or 2 - binary)
- p - precision, how many significant figures are used to represent the mantissa
- $e_{\min/\max}$ - the range of possible exponents / orders of magnitudes with respect to the base β

This results in a finite number of discrete real that can be represented perfectly by the system (further significant figures are rounded off) which are denser towards the minimum order of magnitude. To prevent multiple ways of representing the same number (for example $1 \times 10 = 0.1 \times 10^2$), a **normalized** floating point system requires that $d_0 \neq 0$.

Floating point arithmetic:

1. Convert floating point numbers to the same exponent (ignoring normalized form rules)
2. Perform the operation in binary / whichever base as usual, preserving the common exponent
3. Round off any significant figures lost to precision
4. Normalize ($d_0 \neq 0$) and adjust the exponent accordingly

The total precision / exponent range is dictated by the IEEE standard for a given system, where one bit is usually reserved for signing too. **Double** uses two 32 / 64 bit words (as the name implies), allowing more precision bits and a greater exponent range than **float**; when working on a program where memory usage is non-critical, **double** is greatly preferred.

Rules of thumb:

1. Avoid equality tests involving floating point numbers - these can often return misleading results as many values (such as 1.1) are not perfectly represented in binary
2. Adding numbers of very different orders of magnitude results to lost precision