COMP6771 Lecture Notes (by James Davidson, with additions by Daniel Chen)

*These notes sometimes include small technical asides and additions that are not strictly part of COMP6771. They also don't cover all of the content (e.g. testing using `catch2`).

C++ Basics

`auto` type inference

Try to always use `auto` to get type inference to do the heavy lifting for you. This is a compile time cost, so there is no runtime penalty incurred for doing this.

Be careful though, as it sometimes doesn't work out:

```
auto xs = std::vector<int>{1, 2, 3, 4};

// doesn't quite work, and causes an underflow error that makes this loop infinite
// i is deduced to be of type std::size_t, which is an unsigned type
for (auto i = xs.size(); i >= 0; --i) {
    // ...
}

// of course, you can help type inference out in ambiguous situations like this with type casts,
// but this is just explicitly typing the declaration with extra steps
// one may also argue that this is a contrived example, because reverse iteration
// would be better achieved by iterators
```

`const` keyword

Everything should be `const` by default, unless there is a reason for you to change it. This makes it clearer to the reader of the code that something shouldn't/won't be modified.

Favour *east const* over *west const*, because it tends to make things easier to read.

```
auto const x = 6771; // east const (i.e. right const)
const auto x = 6771; // west const (i.e. left const)

// using east const with some pointers:
int * p;             // p is a mutable pointer to a mutable int
int const * p;       // p is a mutable pointer to a constant int
int * const p;       // p is a constant pointer to a mutable int
int const * const p; // p is a constant pointer to a constant int

// using west const:
const int * p;       // p is a mutable pointer to a constant int
int * const p;       // p is a constant pointer to a mutable int
const int * const p; // p is a constant pointer to a constant int
```

`const` also has benefits in terms of allowing some compiler optimisations, and in multithreaded situations (`const` objects are a lot easier to use in a concurrent setting).

Logical operators in C++20

We can use Python-style `and`, `or`, `not` in place of `&&`, `||` and `!` in the most recent version of C++. Where possible, use these.

Value semantics

The assignment operator has **value/copy semantics**.

```
auto x = std::vector<int>{1, 2, 3};

// x and y are both std::vector<int>, but y is a copy of x
// any changes to x do not manifest in y, and vice versa
auto y = x;
```

Typecasting

Implicit casting is bad, because you never know what's going to happen:

```
auto const i = 42; // int
auto d = 0.0;      // double
d = i;             // 42 gets cast to double, and this happens implicitly
```

Explicit promotions are preferred in this context making your intentions when casting something clear to the compiler that you're doing this, and also to the programmers when they are reading the code.

```
auto const i = 42;                    // int
auto const d = static_cast<double>(i); // i
```

Function syntax

From C++11, there is a new, alternative syntax for functions, called *trailing return type syntax*:

```
// old
int main() {
    std::cout << "Hello World!\n";
}

// new
auto main() -> int {
    std::cout << "Hello World!\n";
}
```

This is nice, because it's more consistent with the notation for lambda expressions.

Function overloading

We can declare functions with the same name, but with different formal parameters:

```
auto f() -> void {}

auto f(int x) -> int {
    return x * x;
}

auto f(double x) -> double {
    return x * x;
}

auto f(int x, int y) -> int {
    return x * y;
}

f(42);
```

When looking for the right function to use, the compiler will check the following in order:

1. Functions matching the name
2. Of those functions, those with the same number of (convertible, if necessary) arguments
3. Of those functions, the best match (in the sense that the type is much better than the others in at least 1 arg)

Overloads should be trivial (e.g. with the same behaviour, just different types). If they are non-trivial, just name the functions differently.

Values and references

Because C++ has value semantics, we have *references* to give us reference semantics. These are kind of like pointers, but with some differences:

- References are an alias for another object and you can use a reference to an object interchangeably with the object itself
- Don't need to do `obj->field` for accessing elements with a reference
- Are non-null
- Are immutable; what they refer to can't change once set

```
auto i = 6771;
auto& j = i; // j is a reference to i
j++;         // actually changes i
```

These by default let you read and write to the thing it references. But you can use `const` to make read-only references:

```
auto i = 6771;
auto const& j = i; // j is a const reference to i; can only read it
j++;               // not allowed
```

If a variable is declared as `const`, all of its references will be read-only:

```
auto const i = 6771;
auto const& j = i; // ok, explicit
auto& k = i;       // still ok, but k will implicitly be a const ref
```

References are typically faster due to avoiding copying (particularly if the values are large in memory).

Pass-by-value and pass-by-reference

```
// pass by value: values are copied into memory being used to hold formal parameters
// (so this swap function doesn't actually work!)
auto swap(int x, int y) -> void {
    auto const tmp = x;
    x = y;
    y = tmp;
}

// pass by reference: formal params are just aliases for the argument, and are
// actually being used (on reads or writes) whenever the formal params are used
// (this swap function *does* work)
auto swap(int& x, int& y) -> void {
    auto const tmp = x;
    x = y;
    y = tmp;
}
```

Pass by reference is useful when the argument has no copy operation and/or the argument is large (so we avoid a potentially expensive copy).

Range-for

A more elegant way of looping over iterable collections:

```
auto xs = std::vector<int>{1, 2, 3, 4};
for (auto const& x : xs) {
    // ...
}
```

We use `const&` because

- Most of the time, you don't want to mutate the thing you're looping over
- Working with references is (usually) faster

Enums

Function mostly the same as they do in C (and other languages):

```
enum class days {
    MONDAY,
    TUESDAY,
    // ...
};

auto const mon = days::MONDAY;
```

STL Containers

(STL = Standard Template Library)

Sequential containers

These organise a finite set of objects into a strict linear arrangement:

- `std::vector` is a dynamically-sized array, and the most common one to use
  - Initial capacity = #. of initial elements that can be stored
  - When the size of the vector reaches capacity, the capacity is doubled
  - Does not automatically shrink capacity; use `v.shrink_to_fit()` to do that
  - Provides two ways to access items:
    - `v.at[i]` does bounds checking, which is more expensive but safer
    - `v[i]` doesn't do bounds checking, which is lightweight but can lead to undefined behaviour
- `std::array` is a more lightweight wrapper for a C-style fixed size array
  - Crucially, while `std::vector` places its memory in the heap, the underlying array in an `std::array` lives in the stack, which can save some system calls
  - The sheer flexibility of `std::vector` means you probably want to use that most of the time unless a specific enough situation occurs for `std::array` to be suitable
- `std::deque` is a double-ended queue
  - Its implementation is a bit more intricate than e.g. a ring buffer
- `std::forward_list` is a singly-linked list
- `std::list` is a doubly-linked list

Most operations on these containers are either $O(1)$ or amortised $O(1)$. The performance of the last 3 can be hindered by cache locality concerns.

Ordered associative containers

These provide fast key-based retrieval of data, but with an order on the elements:

- `std::set` is aset in the mathematical sense
- `std::multiset` is a multiset in the mathematical sense
- `std::map` is a hash table
- `std::multimap` is a hash table with non-unique keys

The ordering here is element sorted order (for `std::map` and `std::multimap`, this is by key), not insertion order. This is achieved by storing them as a search tree (typically, a red-black tree), which gives most operations on them costs of $O(\log{n})$.

A non-default ordering for these containers can be specified by providing a custom element comparator function when constructing it:

```
// this is an int -> int map, but the keys are sorted in descending order
auto m = std::map<int, int, std::greater<int>>{};
```

Unordered associative containers

These provide even faster key-based retrieval of data via hashing, at the cost of any guaranteed ordering of the elements:

- `std::unordered_set` is the unordered version of `std::set`
- `std::unordered_multiset` is the unordered version of `std::multiset`
- `std::unordered_map` is the unordered version of `std::map`
- `std::unordered_multimap` is the unordered version of `std::multimap`

The average complexity of most operations on these is $O(1)$.

Container adapters

These restrict the functionality of an existing container to provide a different set of functionalities:

- `std::stack` is a LIFO stack
- `std::queue` is a FIFO queue
- `std::priority_queue` is a queue where larger elements leave before smaller elements (i.e. it behaves like a max heap)

When declaring container adapters, the underlying sequence container can be specified. For example, by default `std::stack` will use a `std::deque` as its underlying representation.

As with the ordered associative containers, a non-default ordering can be specified for a `std::priority_queue`:

```
// this is a priority queue of ints where smaller elements leave before larger elements
// (i.e. it behaves like a min heap)
auto pq = std::priority_queue<int, std::vector<int>, std::greater<int>>{};
```

Inserters and back inserters

- `std::inserter(c, it)` returns an iterator that allows for the insertion of elements at the location of `it` in the container
- `std::back_inserter(c)` returns an iterator that allows for the insertion of elements at the end of the container

Asides

For certain data structures, there is an `emplace` and a `push/insert/...` function. The difference is that `emplace` constructs an object in-place without using copies or moves by some forwarding magic:

```
auto x = std::stack<MyObj>{};
x.push(MyObj(1, 2, 3)); // creates a temporary copy of the object, then moves it into the stack
x.emplace(1, 2, 3);     // forwards the arguments to the MyObj constructor to create it in the stack in-place
```

This can often be a faster way of doing it as you avoid moves/copies.

Some types cannot be stored in containers:

- References

  ```
  // this is a compile error
  // the full reasoning for this is rather technical, but the crux of the issue
  // is that per the C++ specification, pointers to references are illegal
  // https://eel.is/c++draft/container.requirements#container.reqmts-note-2
  auto vec = std::vector<int&>{};

  // plain pointers can be stored instead, as can a std::reference_wrapper
  // (the latter is a reference replacement that is, among other things, assignable)
  auto vec1 = std::vector<int*>{};
  auto vec2 = std::vector<std::reference_wrapper<int>>{};
  ```

- `const` objects

  ```
  // also a compile error, but again the reasoning for this is slightly technical
  // pre-C++11, the problem was that containers required the inner type to be assignable
  // (which const objects aren't), but this is not true anymore
  // nevertheless, compilers will reject this
  auto vec = std::vector<int const>{};
  ```

STL Iterators

An abstract interface for moving through the items in a container. The implementation of the iterator is defined by whoever is writing that iterator.

Creating iterators

```
container.begin();   // mutable iterator from the start of the container
container.cbegin();  // immutable iterator from the start of the container
container.rbegin();  // mutable reverse iterator from the end of the container
container.crbegin(); // immutable reverse iterator from the end of the container

container.end();     // mutable iterator one past the end of the container
container.cend();    // immutable iterator one past the end of the container
container.rend();    // mutable reverse iterator one before the start of the container
container.crend();   // immutable reverse iterator one before the start of the container
```

Interacting with iterators

Iterators behave a lot like pointers and not references, so you need to use `*` to actually get the item that the iterator is pointing to at any particular time. Dereferencing an `end` iterator is undefined behaviour.

There are two functions for advancing iterators non-linearly (i.e. besides doing `++`):

- `std::advance(it, n)` will advance the iterator `n` steps
- `std::next(it, n)` will create a copy of the iterator advanced `n` steps

In situations where the length of the underlying container may not be known/stored, there is a way to calculate the "distance" between two iterators:

```
auto dist = std::distance(it1, it2);
```

While it makes it a bit more verbose, this approach is useful in certain situations where you may not have a *random-access iterator*. For example, taking the midpoint

of two iterators:

```
// only works if the iterator given back is random-access
auto mid_it_ra = (container.begin() + container.end()) / 2;

// works regardless, though it is a bit longer
auto mid_it = std::next(container.begin(), std::distance(container.begin(), container.end()) / 2);
```

(This, however, is *not* a constant time operation if the iterator is not random-access!)

Types of iterators

- Input iterator: read-only iterator that can be incremented or compared for (in)equality
- Output iterator: write-only iterator that can be incremented or compared for (in)equality
- Forward iterator: like an input/output iterator but you can both read and write
- Bidirectional iterator: like a forward iterator but you can decrement too
- Random-access iterator: most general kind of iterator, which, in addition to all of the previously listed features, provides
    - Relational comparisons (e.g. `it1 < it2`)
    - Iterator arithmetic (e.g. `it1 + it2`)
    - Subscript access to values (e.g. `it[k]`, which is just `*(it + k)`)

In general read/write situations, we have forward iterators $\subset$ bidirectional iterators $\subset$ random-access iterators.

Different STL containers provide different iterator types:

- `std::vector`, `std::deque` and `std::array` give random-access iterators
- `std::list`, `std::(multi)set` and `std::(multi)map` give bidirectional iterators
- `std::forward_list`, `std::unordered_(multi)set` and `std::unordered_(multi)map` give forward iterators

Container adapters do not provide iterators.

In C++20, there are also contiguous iterators, which are random iterators that guarantee contiguity of the underlying elements in memory.

Asides

When using a `const` iterator in a loop for example, you don't make the actual iterator variable `const` explicitly:

```
for (auto const it = c.begin(); it != c.end(); ++it) {
    // doesn't work, needs to be auto it = ...
}
```

When working with something like a `map`, if you want to look up whether an item is in the map and then use that item if it does exist, it is often quicker to work with the `.find()` method, which will give back an iterator:

```
// iterator method: one lookup and can access the item at that key via the iterator
// note: the iterator is to a key-value std::pair, hence the use of ->
auto it = map.find(key);
if (it != map.end()) {
    auto v = it->second;
}

// since C++17, you can actually put an init statement inside an if, which is nice if
// you have no use for the iterator variable after the body of the if is executed
if (auto it = map.find(key); it != map.end()) {
    auto v = it->second;
}

// since C++20, there is a map.contains() method to check existence,
// but accessing the value afterwards requires a second lookup
if (map.contains(key)) {
    auto v = map.at(key);
}
```

STL Algorithms

The `<algorithm>` library gives some nice standardised functions for common tasks to cut down on boilerplate code that the programmer has to write.

These algorithms work on iterators to containers rather than containers directly so as to be most portable with different containers.

Map and reduce equivalents

```
// this is map, which places the mapped contents into a destination container
// it is up to the programmer to make sure that destination container is big enough!
std::transform(src_begin, src_end, dest_begin, func);

// this is basically a non-returning version of map
// the func should be void, because its return type is ignored
// to mutate, make sure func takes references in the argument, and change via assignment
std::for_each(begin, end, func);

// this is reduce
// function takes (accumulator, value) as arguments (in that order)
auto result = std::accumulate(begin, end, initial_value, func);
```

For more functional-esque tools, see the `<functional>` header (similar to Python's `operator` and `functools` modules).

Lambda expressions

To specify an unnamed function, we can use a lambda expression:

```
[capture] (args) -> ret_type {
    body
}
```

The return type is optional to specify here.

The capture part is necessary because, by default, the lambda does not get access to its surrounding scope. Things you want to explicitly add to the scope of the lambda should be given in a comma-separated list inside the square brackets.

A (by default `const`) copy of each captured variable is made, with value initialised to that of the variable in the outer scope at the point at which the lambda is defined:

```
auto n = 6771;
auto f = [n] (int x) -> int {
    return x + n;
};
n++;
f(1); // still gives 6772 even though n has changed after defining the lambda
```

To make the local copies of captured variables mutable, we can add `mutable` after the parameter list:

```
[capture] (args) mutable -> ret_type {
    body
}
```

A lambda capture can contain variables with initialisers (and these may also shadow variables of the same name from an outer scope), but to modify their values, the lambda expression must still be `mutable`:

```
int i = 2;
auto f = [i = 0] (int x) mutable -> int {
    return (i++) * x;
};
f(1); // 0
f(1); // 1
i;    // 2
```

To mutate the thing being captured by the lambda after it has executed, we can capture by reference: `[&var]`.

If we want everything mentioned in the body of the lambda expression to be captured by value or reference automatically without having to list them all, use `[=]` or `[&]` as the capture expression respectively.

Asides

Lambda expresions are, under the hood, actually implemented as anonymous *functors*. A more critical difference between a functor from a function is that functors can have state (e.g. the captured variables in these examples).

Class Types

Construction

A class can typically offer more than one way of creating a new instance:

```
// default construction: calls the no-arg constructor
std::vector<int> v11;          // v11 == empty vector
auto v12 = std::vector<int>{}; // v12 == empty vector

// copy constructors
auto v2 = std::vector<int>{v11.begin(), v11.end()}; // v2 == copy of v11
auto v3 = std::vector<int>{v2};                     // v3 == copy of v2

// initialiser list constructor
auto v4 = std::vector<int>{5, 2}; // == vector with contents [5, 2]

// count + value constructor
auto v5 = std::vector<int>(5, 2); // == vector with contents [2, 2, 2, 2, 2]
```

Note that, aside from the last example, braces `{}` were used instead of parentheses `()` when invoking the constructor. This is called *uniform initialisation*, which is preferred to use over *direct initialisation*, since it is more strict with narrowing conversions (e.g. `double` to `int`) that may implicitly happen with constructor arguments.

Namespaces

Namespaces allow us to group things that belong together. They're also used to prevent similarly-named things from clashing.

```
namespace my_namespace {
auto x = 6771;
} // namespace my_namespace

// refer to this in later code as my_namespace::x;
```

It's customary to not indent the namespace block itself, but its contents have its own indentation.

They can be nested, but we prefer top-level namespaces to multi-tier:

```
namespace x {
namespace y {
auto z = 6771;
} // namespace y
} // namespace x

// refer to this in later code as x::y::z

// or we could do the following to reduce nesting, which is cleaner
namespace x::y {
auto z = 6771;
} // namespace x::y
```

They can be anonymous (i.e. unnamed), which can be used to simulate the effect of `static` functions in C. These are local to the file in which they are defined:

```
namespace {
auto f(int x) -> int {
    return x + 1;
}
} // namespace

// refer to the functions in such a namespace just using their names
```

We can give namespaces new names, e.g. `namespace chrono = std::chrono;`.

We always fully-qualify things (e.g. STL containers) to avoid counterintuitive behaviour with overloading resolution. This means that `using` directives such as `using namespace std;` are frowned upon.

Aside

In addition to namespace aliases, another alternative to shortening long types are type aliases:

```
// type aliases can be local to a scope (e.g. a function or class),
// or global for a file (including files which #include it)
using map_of_maps = std::unordered_map<int, std::unordered_map<int, int>>;
```

There are some instances where `using` directives *do* make sense:

```
auto print_time_fact() -> void {
    // this using directive is local only to the block scope of print_time_fact,
    // and it's appropriate to do this here, because time literals would be very,
    // very painful to use otherwise
    using namespace std::chrono_literals;
    std::cout << "There are "
              << std::chrono::seconds(6771m).count()
              << " seconds in 6771 minutes\n";
}
```

OOP in C++

```
class foo {
    // until an accessibility modifier is encountered, everything in a class is assumed private
    int a;
    void f();

// members accessible by everyone
public:
    foo();

// members accessible by members, friends and subclasses
protected:
    int x;

// members accessible by members and friends
private:
    int y;
    void z();

// can have multiple sections of the same kind
public:
    int w;
}

struct foo {
    // like a class, but these are public by default!
    int a;
    int b;
    int c;

// but it can have private stuff too
private:
    void f();
}
```

As with namespaces, it is customary not to indent a class block.

By default, members of a class are *private*. The **only** difference between a `struct` and a `class` is that all members of a `struct` are *public* by default. We will almost always use a `class` unless we essentially just want a data class.

There is a notion of `this` (i.e. `self` in Python), which in a class-defined function call is always a pointer to the class object which called it. However we prefer to just suffix private/internal members with an underscore.

Class scope

It's common to declare a class with its method signatures in a header file, then implement those methods in another file. However, the implementation must scope the class when writing those implementations:

```
// in Foo.h
class Foo {
public:
    Foo();
    ~Foo();
    void f();
}
```

```
// in Foo.cpp
#include "Foo.h"

Foo::Foo() {
    // ...
}

Foo::~Foo() {
    // ...
}

void Foo::f() {
    // ...
}
```

Constructors

Constructors may specify an *initialiser list*, which gives values to data members in order of their member declaration in the class itself:

```
class MyClass {
public:
    MyClass(int i, std::vector<int> j) : i_{i}, j_{j} {
        // ...
    }
private:
    int i_;
    std::vector<int> j_;
}
```

Crucially, this happens *before* the constructor body is actually executed.

When initialising an object, the following order is used:

```
for each data member in declaration order
    if it has a used definition initialiser
        initialise it using the used defined initialiser
    else if it is of a built-in type
        do nothing (leave it as whatever)
    else
        initialise it using its default constructor
```

In other words, initialisation happens for all data members before the body is called, making so-called uniform initialisation more efficient than setting things in the constructor body (since those values are initialised first anyway, perhaps just to default values). You may as well use an initialiser list to just make those initial values meaningful.

Delegating constructors

Constructors can call other constructors, possibly to set default values:

```
class MyClass {
public:
    MyClass(int i, std::vector<int> j) : i_{i}, j_{j} {}
    MyClass(std::vector<int> j) : MyClass(6771, j) {};
private:
    int i_;
    std::vector<int> j_;
}
```

Destructors

Are functions that are called in the moment before an object goes out of scope, which can be useful for cleaning up used resources (e.g. any pointers, opened files or locks owned by the object). They should not throw exceptions.

```
class MyClass {
    ~MyClass() noexcept;
}

MyClass::~MyClass() noexcept {
    // do destruction, e.g. closing a file
}
```

Explicit initialisation

By default, unary constructors can do implicit type conversion from the parameter to the class:

```
class Age {
public:
    Age(int age): age_{age} {}
private:
    int age_;
}

// explicit construction
Age a1{12};
auto a2 = age{12};

// implicit construction
Age a = 12;
```

Sometimes you want this, other times you don't, because implicit type conversions are generally not liked. To prevent this and force people to do the explicit way, use the `explicit` keyword:

```
class Age {
public:
    explicit Age(int age): age_{age} {}
private:
    int age_;
}

// explicit construction still works
Age a1{12};
auto a2 = age{12};

// implicit construction is now an error
// Age a = 12;
```

`const` objects and member functions

By default, member functions are only callable by non-`const` objects. Only member functions marked with `const` at the end of the function may be called on `const` objects:

```
class Person {
public:
    person(std::string const& name) : name_{name} {}

    // only callable by non-const Person objects
    auto set_name(std::string const& name) -> void {
        name_ = name;
    }

    // callable by all objects
    // it is a compiler error to modify members in such a function that are not
    // declared as mutable; it's rare to want to ever make members mutable,
    // but they do have their use cases sometimes (e.g. a cache)
    auto get_name() const -> std::string const& {
        return name_;
    }
private:
    mutable int age;
    std::string name_;
}
```

Static data members and member functions

Belong to every instance of a class:

```
class MyClass {
public:
    static std::string const x;
    static void f();
}

// member function may be called as MyClass::f()
```

Static data members in general can't be initialised in the class itself, but must be done elsewhere:

```
auto MyClass::x = "abcd";
```

Special member functions, `default` and `delete`

By default, the compiler will *synthesise* or create some *special member functions* for you. Two examples are

- If no constructors are given, then a default no-arg constructor will be created for you
- A copy constructor that allows one to construct an object as a copy of an existing one

To signal to the compiler that you do want its synthesised constructors (e.g. the default constructor), use the keyword `default`. To signal to the compiler that you *don't* want one of its synthesised constructors (e.g. the copy constructor), use the `delete` keyword.

```
class MyClass {
public:
    MyClass() = default; // generates the default constructor
    MyClass(int i) i_{i} {}
    MyClass(MyClass const& mc) = delete; // don't generate the copy constructor
private:
    int i_;
}
```

Operator Overloading

In C++, all operators (like `<`, `==`, `[]`) are functions and can be overloaded to work with custom classes. Each operator is prefixed by the word `operator` (e.g. the `==` operator is `operator==` as a function). In general, however, it only makes sense to create an overload for an operator of some type if, when used with that type, it has a single, obvious meaning.

Type	Operator(s)	Member or friend?
I/O	`>>`, `<<`	Friend
Arithmetic	`+`, `-`, `*`, `/`	Friend
Comparison	`>`, `<`, `>=`, `<=`, `==`, `!=`	Friend
Assignment	`=`	Member (non-`const`)
Compound assignment	`+=`, `-=`, `*=`, `/=`	Member (non-`const`)
Subscript	`[]`	Member (`const` and non-`const`)
Increment/decrement	`++`, `--`	Member (non-`const`)
Dereference	`->`, `*`	Member (non-`const`)
Function call	`()`	Member

Friendship

Making a non-member function a friend of a class allows it to access otherwise private internal member fields. This obviously breaks abstraction and should be avoided wherever possible, but it does have its uses:

- Operator overloading
- Allowing member functions of related classes (e.g. iterators) to access class internals

---

Custom Iterators

Iterator invalidation

When we modify a container, this may affect iterators to that container. For example, if we are continually moving the endpoint of a container by inserting/deleting elements from it, it is likely not the case that an old `end()` iterator is valid anymore. This is the concept of *iterator invalidation*: some operations may render existing iterators invalid, such that continued use of these iterators is now undefined behaviour.

Which operations do and don't invalidate iterators should be specified by the operations of the container. Some operations on some containers may only invalid some iterators (e.g. it might only invalidate the past-the-end `end()` iterator).

Iterator traits

Each iterator has certain properties:

- Category: is it an input/output, forward, bidirectional or random-access iterator?
- Value type: what is the type of the element that the iterator points to?
- Reference type: what is the type of references to elements that the iterator points to?
- Pointer type: what is the type of pointers to elements that the iterator points to?
- Difference type: what is the type that results upon subtraction of iterators?

Building iterators

An iterator, at minimum, must look like this:

```
#include <iterator>

template <typename T>
class my_iterator {
public:
    using iterator_category = std::forward_iterator_tag; // or some other iterator category
    using value_type = T;
    using reference = T&;
    using pointer = T*;
    using difference_type = int;

    my_iterator& operator++();
    my_iterator operator++(int) {
        auto copy{*this};
        ++(*this);
        return copy;
    }

    reference operator*() const;

    // not strictly required, but it's nice to have
    pointer operator->() const {
        return &(operator*());
    }

    // in C++20, operator!= is automatically derived for you from operator==
    friend bool operator==(const my_iterator& lhs, const my_iterator& rhs) {
        // ...
    }
};
```

For bidirectional iterators, one also needs to provide prefix and postfix `operator--`.

To allow a custom container to be used with STL functions that accept iterators, all one needs to do is provide implementations of `begin()`, `end()`, `cbegin()` and `cend()`. Within the container, one should also specify some iterator types:

```
using iterator = // your iterator type here
using const_iterator = // your const iterator type here
```

If you have a bidirectional iterator already, you can get reverse iterators for free by using the `reverse_iterator` and `const_reverse_iterator` iterator adaptors.

Exceptions

Exceptions are a runtime mechanism for signifying exceptional circumstances during code exectution. Exception handling is the process of managing exceptions that are raised rather than causing the program to crash.

Exception objects

All C++ exceptions are objects which derive from (i.e. are subclasses of) `std::exception`. As such, we

- throw exceptions by value
- catch exceptions by `const` reference

Exception control flow

```
try {
    // some code
} catch (exception1_t const& e1) {
    // some logging
} catch (exception2_t const& e2) {
    // some more logging
} catch (...) {
    // logging for any exception other than the previous 2
}
```

Rethrow

```
try {
    try {
        // some code
    } catch (exception_t const& e) {
        // some logging

        // exception is rethrown for handling by the next try/catch layer
        throw e;
    }
} catch (exception_t const& e) {
    // some further logging
}
```

No-throw exception safety (failure transparency)

An operation that is guaranteed to never throw an unhandled exception provides no-throw exception safety. While exceptions may occur, they are handled internally. Some examples of such operations:

- Closing files
- Freeing memory
- Move constructors and move assignments
- Trivial stack object creation

Strong exception safety (commit or rollback)

An operation that may fail, but without leaving visible effects (e.g. no modifications to the object's state) provides strong exception safety. This is the most common type of exception safety offered by C++ functions.

To achieve this, first perform all throwing operations that don't modify internal state before doing irreversible, non-throwing operations.

Basic exception safety (no-leak guarantee)

An operation that may fail and cause side effects, but

- respects class invariants
- does not leak resources
- corrupt data on exception provides basic exception safety. Objects are afterward left in a *valid but unspecified state*, as there is no telling the extent to which the side effects of partial execution have modified things.

No exception safety

Operations that make no guarantees regarding exceptions provide no exception safety. This is often bad C++ code and should be avoided at all costs (especially since wrapping resources and attaching lifetimes to them can give at least basic exception safety).

`noexcept`

Functions marked as `noexcept` are understood to not throw unhandled exceptions (but doesn't technically prevent them from doing so). STL functions can operate more efficiently on `noexcept` functions.

Resource Management

Long lifetimes

There are three ways to make an object's lifetime outlive that of its defining scope:

- Returning out from a function via copy
- Returning out from a function via reference
    - The object itself must always outlive the reference, so references to local function variables can result in undefined behaviour!
- Returning out from a function as a heap resource

Heap allocation via `new` and `delete`

In C++, the equivalents of `malloc` and `free` are `new` and `delete`. These call the constructors/destructors of a class to create/delete instances of that class.

```
int* i = new int{6771};
delete i;

std::vector<int>* v = new std::vector<int>{1, 2, 3};
delete v;

int *l = new int[123];
delete[] l; // calls destructor on each elem first before deallocing the array
```

Since the heap is global unlike local function stacks, this means they outlive their defining scopes.

Destructors

When a non-reference object goes out of scope, the destructor of that object is called, which frees any underlying heap resources that may be under its control. Examples of where this might be useful:

- Freeing pointers
- Closing open files
- Releasing locks

The process during which these destructor calls are inserted at the end of a scope is called *stack unwinding*.

Rule of 5/0

When thinking about resource management for a class, there are 5 operations to keep in mind:

- Destructor
- Copy constructor and copy assignment
- Move constructor and move assignment

The *rule of 5* states that if a class defines custom implementations any of the 5 listed operations, then it should also provide custom implementations of the other 4. The reasoning behind this is that if the default behaviour isn't sufficient for one of them, then it likely isn't sufficient for the others too. (Prior to C++11, this was the rule of 3, since copy/move assignment didn't exist then.)

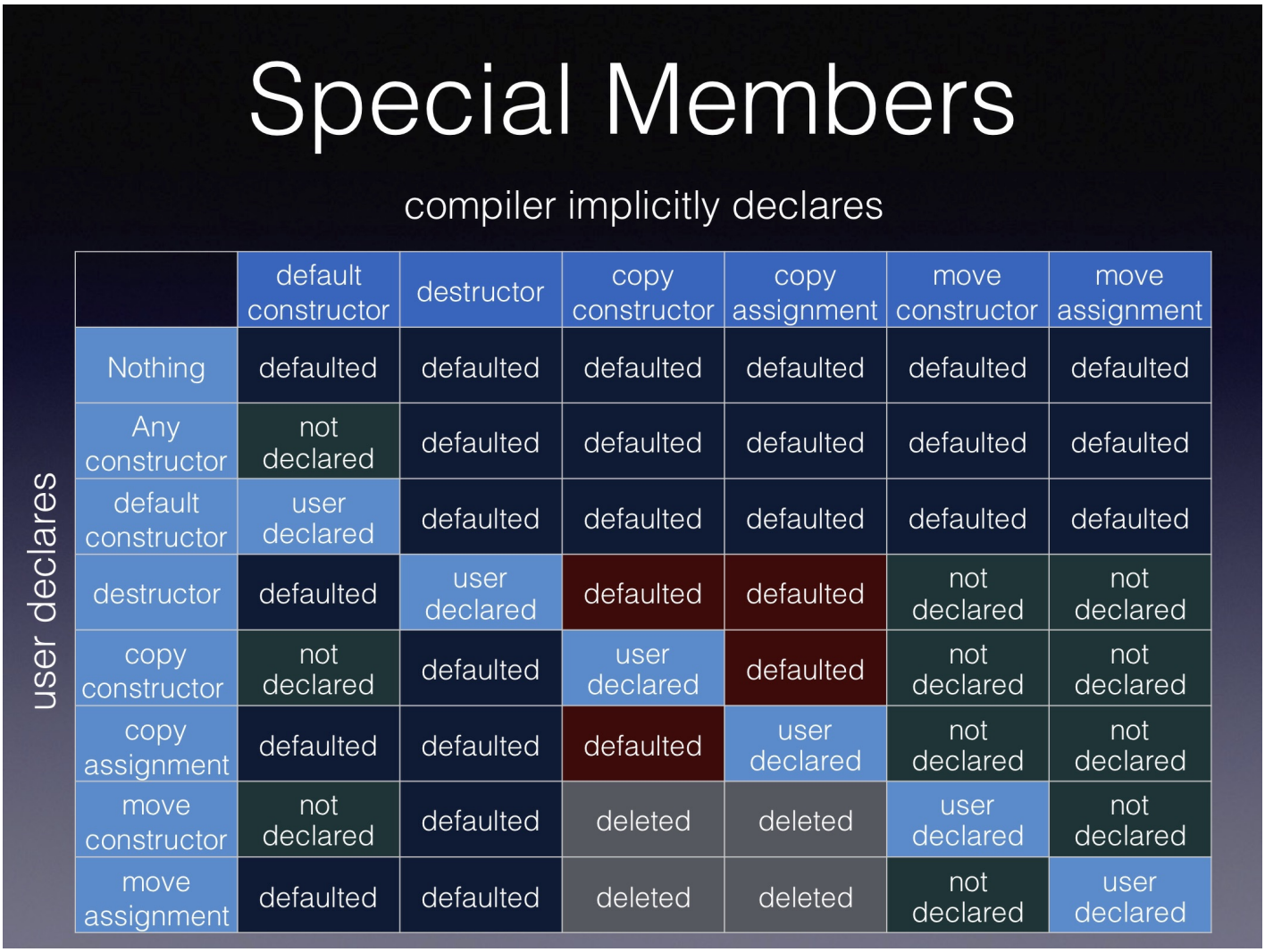
The *rule of 0* states that a class requiring managed resources should either

- take full responsibility its resources by declaring and implementing all 5 operations
- declare none of these operations and rely on the default implementations by instead using types that *do* internally manage each resource (through their own implementations of the 5 operations).

```
class cstring {
public:
    // rule of 5: cstring takes full responsibility over its resources
    ~cstring() { delete[] p; }
    cstring(cstring const&);
    cstring(cstring&&);
    cstring operator=(cstring const&);
    cstring operator=(cstring&&);
private:
    char* p;
}

class person {
    // rule of 0: none of the 5 operations are declared and are implicitly defaulted
    // (this is now effectively just a dataclass)
    cstring name;
    int age;
}
```

Implicit stuff table added by Daniel



Custom copy constructors

Because the default behaviour of a copy constructor is to perform member-wise shallow copies of data members, any pointer resources (e.g. heap arrays) will refer to the same object in both object copies (i.e. changes in one object to this data member reflect in the other). Even worse, during destruction, such resources will be double-freed. So, when writing a copy constructor, make sure to do deep copies:

```
my_vec::operator=(my_vec const& mv)
: data_{new int[mv.size_]}
, size_{mv.size_}
{
    std::copy(mv.data_, mv.data_ + mv.size_, data_);
}
```

Custom copy assignment operators

The copy-and-swap idiom is most useful for implementing copy assignment correctly:

```
my_vec::my_vec(mv_vec const& mv) {
    // use the copy constructor to create a copy of the object,
    // then swap out its internals with this object
    my_vec(mv).swap(*this);
    return *this;
}

void my_vec::swap(my_vec& mv) {
    std::swap(data_, mv.data_);
    std::swap(size_, mv.size_);
}
```

Value categories: lvalues and rvalues

An *lvalue* is an expression that is an object reference, which is to say that it refers to an object with a defined address in memory. On the other hand, an *rvalue* is anything that isn't an lvalue. Things which are lvalues include variable names, and things that are rvalues are non-string object literals (e.g. integers) and return values of functions.

```
int a = 3;     //  a = lvalue, 3 = rvalue
int b = f(a);  //  b = lvalue, f(a) = rvalue
int c = b;     //  a, b = lvalue
int d = a + 1; // d = lvalue, a + 1 = rvalue
```

These are the two main *value categories*, but in reality, it is not a strict dichotomy.

Loosely speaking, `std::move` can be used to turn an lvalue into an rvalue. (This is a bit of a lie: in reality, what is created is instead an *xvalue*, or expiring value.)

Move constructor

Rather than creating new instances of a class by copying, we could instead construct it from the internals of another object by moving:

```
// std::capacity(obj, val) replaces the contents of obj by val, and returns the old value
my_vec::my_vec(my_vec&& orig) noexcept
: data_{std::exchange(orig.data_, nullptr)}
, size_{std::exchange(orig.size_, 0)}
, capacity_{std::exchange(orig.capacity_, 0)} {}
```

These should always be marked `noexcept`, as throwing move constructors are very rare, and by specifying a function as `noexcept`, the compiler can perform many optimisations to improve performance (namely, it doesn't have to worry about generating exception code).

In effect, a new object is created by "stealing" the resources of the moved object. Afterwards, the moved object should be left in a *valid but indeterminate/unspecified state*. Validity is dependent on the exact type being worked with (i.e. what constitutes a valid object of one type depends on the requirements for values of that type), however being in an unspecified state means that you cannot determine what its internal state may look like (i.e. you might not be able to say for sure what a specified member field's value will be).

It is bad practice to use a moved-from object after move construction/assignment, and often constitutes undefined behaviour. A well-configured compiler will warn about this.

RAII

Resource Acquisition Is Initialisation is a C++ programming technique in which resources (i.e. heap objects) are encapsulated inside objects, therefore binding the life cycle of an acquired resource to the lifetime of that object. We *acquire* the resource in the constructor, and we *release* the resources in the destructor.

Every resource should be owned by one of the following:

- Another resource (e.g. smart pointer, data member)
- A named rsource on the stack
- A nameless temporary variable

Smart pointers

In C++11 and onwards, smart pointers are a way of wrapping unnamed heap objects (i.e. raw pointers) in named stack objects so that the lifetime of that heap object can be more easily managed (in keeping with the spirit of RAII).

Unique pointers (`std::unique_ptr`)

A unique pointer is an abstraction that takes *unique* ownership of a heap resource. When constructed with an initial heap object to manage, the unique pointer is the sole object who owns the managed resource, with all other access to it coming from raw pointers acting as observers. This forms the common usage pattern with unique pointers: dominion over the resource is established by the unique pointer, which is then held by some object, and any additional references to the underlying heap value are via raw observer pointers.

```
// my_up now owns the heap resources associated with the string "hello"
auto up = std::make_unique<std::string>("hello");

// we use make_unique when constructing them, as it's better than the alternative,
// i.e. explicit use of new (prone to issues regarding use of unnamed temporaries)
auto up = std::unique_ptr<std::string>(new std::string("hello"));

// we can get a raw pointer to the underlying value, a so-called observer
// p_raw will be of type std::string*
auto p = up.get();

// we can access the actual value pointed to using operator* and operator->
auto v = *up;

// we can also relinquish ownership of the resource, and other stuff too
// r will now be a std::string*
auto r = up.release();
```

Since they are unique, these types of pointers are not copy-constructible/assignable (i.e. these are explicitly `delete`d in their implementations).

Once a unique pointer goes out of scope, its destructor is called. Naturally, when this destructor is called, its managed heap resource is destroyed. A consequence is that any observer pointers to a unique pointer that goes out of scope will thus point to garbage memory, so some care has to be taken when using an observer pointer.

Shared pointers (`std::shared_ptr`)

A shared pointer is an abstraction that allows ownership of an object to be shared across multiple pointers, instead of uniquely owned by one pointer. This essentially acts as a reference-counted pointer, in that the underlying heap resource is only ever destroyed once the number of shared pointers which point to it hits 0, allowing some pointers to come and go without leading to the demise of that heap object. This reference count is updated atomically to allow use in concurrent situations. (There is no non-atomic built-in alternative, unlike e.g. Rust's `Rc` and `Arc`.)

A shared pointer is used in much the same way as a unique pointer, except that you obviously cannot relinquish ownership of the resource, and that we can view the active reference count for the heap resource.

```
// sp1 is now one of the shared pointers to the string "hello"
auto sp1 = std::make_shared<std::string>("hello");

// can access the underlying value by operator*
auto v = *sp1;

{
    // we make more shared pointers by copy-constructing another shared pointer
    // this adds 1 to the reference count of all shared pointers to that resource
    auto sp2 = sp1;
    auto sp3 = sp2;

    // we can view the reference count at any time
    auto rc = sp1.use_count(); // = sp2.use_count() = sp3.use_count() too
}

// sp2 and sp3 are now destroyed, but the heap object remains until sp1 is destroyed
```

Because not all accesses of a value necessarily need to be responsible for the object itself, we can create an analogue of "observers" by using weak pointers. These do not contribute to the reference count of that heap resource, but if usage of the resource is required (perhaps only temporarily), it must be converted to a shared pointer. Otherwise, all one can do with a unique pointer is check whether the shared pointer's managed resource is there or not.

```
// wp is now a weak pointer to sp1
auto wp = std::weak_ptr<std::string>(sp1);

// if pointer is expired, then the heap resource is gone
if (wp.expired()) {
    // nothing we can do with wp now safely
}

// on the other hand, if it does exist, to get access to it, we must get a "lock" on it first
// (i.e. get a shared pointer to the resource)
else {
    auto sp = wp.lock();
    // free to use the resource safely now, and this reference will be cleaned up
    // when sp goes out of scope
}

// from a weak pointer, you can view the reference count
auto rc = wp.use_count();
```

When to use unique and shared pointers

It's almost always the case that you want unique ownership over the object instead of shared ownership, so it makes sense to use unique pointers over shared pointers.

You should use shared pointers if, for example:

- An object has multiple owners, and it's unclear as to which one will be the longest-lived (e.g. a multithreaded situation needing access to a common resource, where it may not be known which thread exits last)
- You need safe temporary access to an object, which is not guaranteed for observing raw pointers of a unique pointer

These situations are rare in simple use cases, though.

Smart pointers and partial construction

If an exception is thrown in a constructor, then only some of its subobjects (i.e. fields) are fully constructed. The C++ standard states that only destructors for these fully constructed subobjects are called, but crucially the constructor of the object itself is not called.

When working with raw pointers, this is a problem, because the destructor of a pointer does nothing, leading to memory leaks. By using smart pointers, the leak is avoided, as upon subobject destruction, the destructor of a smart pointer is called, which frees underlying memory (at least in the case of a unique pointer).

This implies that, as a general rule of thumb, it is better to manage several wrappers around individual objects which each are responsible/have ownership over that singular resource.

Dynamic Polymorphism

Polymorphism is the provision of a single interface to entities of different types. When this occurs at runtime, we call it dynamic polymorphism. In contrast to some other language, this is (at least for the most part) without performance penalty, in keeping with the C++ ethos of "you don't pay for what you don't use".

Inheritance

```
class base {
public:
    int x;
    auto foo() -> void;

// as in other languages, protected members are accessible only to objects in a subclass hierarchy
// (i.e. base itself, and any class which derives from base)
protected:
    int y;
    auto bar() -> void;

private:
    int z;
    auto baz() -> void;
};

// note the access specifier!
class derived : public base {
public:
    // we can 'override' functions like usual
    auto foo() -> void;
};
```

The access specifier above dictates the maximum level of accessibility of things inherited from the base class:

```
// public inheritance essentially changes nothing about accessibility
class derived : public base {
    // ...
};

// with protected inheritance, the inherited public members of base have their accessibility
// capped at protected, i.e. no longer part of derived's public interface
class derived : protected base {
    // ...
};

// with private inheritance, the inherited public and protected members have their accessibility
// capped at private, i.e. only accessible from within derived itself
class derived : private base {
    // ...
};

// the access specifier is optional, and if it is left out, it's just private inheritance
class derived : base {
    // ...
};
```

Unless you have a good reason, we typically do public inheritance.

The member variable layout in memory of a derived class is that of contiguous subobjects:

```
|^^^^^^^^^^^^^^^^^^^^^^^^^^^^^|
| base subobject:             |
| - member variables          |
|-----------------------------|
| derived subobject:          |
| - non-base member variables |
|_____________________________|
```

Object slicing problem

Since the objects of a derived class may be larger than objects of its base class, it's unclear to the compiler how much space one would need to allocate in a situation like this:

```
// since a class could derive from base, how much space do we allocate on the stack
// to hold the obj argument?
auto do_something(base obj) {
    obj.say_hi();
}
```

The only sensible answer is to allocate enough size for a base class object. However, a consequence of this is that when a derived class object is passed by value as an argument to `do_something`, the copy of the object residing in the memory allocated for the `obj` argument only contains the data of the base class subobject. This is the *object slicing problem*: the additional memory usually contained in a derived class object has been "sliced off" in the copy.

Passing by reference (i.e. references or raw pointers) will avoid object slicing, so we always prefer this when dealing with inheritance hierarchies.

Dynamic binding

When passing base class objects by reference, C++ uses, by default, the base class implementation of functions, even if they have been overriden in a derived class:

```
class base {
public:
    auto say_hi() -> void {
        std::cout << "Hi from the base class\n";
    }

    auto say_bye() -> void {
        std::cout << "Bye!\n";
    }
};

class derived : public base {
public:
    auto say_hi() -> void {
        std::cout << "Hi from the derived class\n";
    }
};

auto do_something(base& obj) {
    obj.say_hi();
}

do_something(base{});    // prints "Hi from the base class" (fine)
do_something(derived{}); // prints "Hi from the base class" (weird)
```

This is mostly a performance consideration; this default is easily determined at compile time, since derived class object references can be *statically bound* to base class object references.

However, C++ can be forced to use *dynamic binding* to determine the right override to pick at runtime using *virtual functions*:

```
class base {
public:
    // the virtual keyword indicates that this function may be overriden in a derived class,
    // and so C++ must now put in the effort to determine which one to call from the context
    virtual auto say_hi() -> void {
        std::cout << "Hi from the base class\n";
    }

    virtual auto say_what() -> void {
        std::cout << "What?\n";
    }

    // this function is non-virtual, i.e. normal
    auto say_bye() -> void {
        std::cout << "Bye!\n";
    }
};

class derived : public base {
public:
    // the override keyword indicates that it is an override of a base class function
    auto say_hi() -> void override {
        std::cout << "Hi from the derived class\n";
    }

    // the subclass doesn't have to override *every* virtual from the base class
};

auto do_something(base& obj) {
    obj.say_hi();
}

do_something(base{});    // prints "Hi from the base class"
do_something(derived{}); // prints "Hi from the derived class"
```

The `override` keyword is technically optional, but there are benefits to using it:

- If there is no virtual base class function of the same name, using `override` will pick this up at compile-time
- It's less ambiguous and error-prone for programmers

Virtual functions rely an under-the-hood abstraction called a *vtable*. Each class has its own vtable, stored in the data segment, consisting of an array of function pointers to the definition of each virtual function.

```
in the code segment, we have
    base::say_hi() { /* ... */ }
    derived::say_hi() { /* ... */ }
    base::say_what() { /* ... */ }

in the data segment, we have
    base_vtable:    [<pointer to base::say_hi>,    <pointer to base::say_what>]
    derived_vtable: [<pointer to derived::say_hi>, <pointer to base::say_what>]
```

If a class has a non-empty vtable, each object of that class internally also holds a pointer to the vtable of its class to aid in dynamic binding. When a virtual function is called on such an object passed by reference (i.e. either by reference or pointer type), C++ will

- Follow the object's vtable pointer
- Use offset arithmetic (specific to each virtual function) to find the correct function pointer in the vtable
- Follow this function pointer to get to the definition of the virtual function and call it

This involves additional runtime overhead beyond a normal function call (more machine instructions are required), and also incurs indirect memory accesses (which can have poor cache performance), so is undesirable in performance-sensitive code. Sometimes, compilers can devirtualise virtual function calls to avoid this extra work.

TODO: default args and virtuals

Finality

We can specify that a virtual function in a derived class will not be further overriden by any of its own derived classes, i.e. will not be virtual further down the inheritance hierarchy:

```
class base {
public:
    virtual auto say_hi() -> void {
        // ...
    }
};

class derived : public base {
public:
    // the final keyword indicates that no class which inherits from derived will
    // also override this function, and a compile error will be generated if
    // one attempts to do so
    auto say_hi() -> void final override {
        // ...
    }
};

// even though obj could refer to objects of a class which inherit from derived,
// they won't override say_hi by finality, so we can just do static binding here rather
// than dynamic binding, which eliminates the performance overhead of the vtable
auto do_something(derived& obj) {
    obj.say_hi();
}
```

Finality can also be applied to classes themselves, preventing other classes deriving from it:

```
class base final {
    // ...
};

// this produces an error at compile time now
class derived : public base {
    // ...
};
```

Pure virtual functions and abstract classes

A virtual function is considered to be *pure virtual* if it has no corresponding implementation in that class:

```
class base {
public:
    // the pure specifier "= 0" indicates that this base class virtual function
    // doesn't come with an implementation of say_hi,
    // so in effect this just acts as an interface
    virtual auto say_hi() -> void = 0;

    virtual auto say_bye() -> void {
        std::cout << "Bye!\n";
    };
};

class derived : public base {
public:
    auto say_hi() -> void override {
        // we now have to implement it in any derived classes
        // say_hi becomes non-pure virtual for any further subclasses of derived
    }
};

class intermediate : public base {
public:
    // we can also have pure virtual overrides of non-pure virtual functions
    // anything which derives from intermediate must implement say_bye
    auto say_bye() -> void override = 0;
};
```

If a class has at least one pure virtual member function, then objects of that class *cannot* be constructed. Moreover, functions cannot accept or return objects of such a class type by value, only by reference.

Pure virtual functions allow us to mimic the behaviour of *abstract classes* from other OOP languages.

OOP type theory

TODO: covariance, contravariance, ...

Polymorphism and construction

To avoid the object slicing problem, we must use pointers to store polymorphic objects in, say, a container:

```
// this doesn't work, because all of the contents of the vector are stored inline,
// which introduces object slicing
auto objs = std::vector<base>{};
objs.push_back(base{});
objs.push_back(derived{});

// we know we can't store references, so we must do pointers (raw or smart)
auto objs = std::vector<std::unique_ptr<base>>{};
objs.push_back(std::make_unique<base>());
objs.push_back(std::make_unique<derived>());

// TODO: there is, supposedly, a subtle problem with this code,
// but i don't see it right now
```

Since derived class objects contain base class subobjects, derived classes must call a base class constructor:

```
class base {
public:
    base(int x) : x_{x} {}
private:
    int x_;
};

class derived : public base {
public:
    // derived constructor calls base constructor
    // if it doesn't, then the default constructor of base is implicitly called
    derived(int x, int y) : base(x), y_{y} {}
private:
    int y_;
};
```

Initialisation of the base subobject within a derived class object cannot be done within the derived class, even for protected members.

TODO: 6:56

Polymorphism and destruction

TODO

Templates

Templates are a form of *static* polymorphism in C++, i.e. compile time polymorphism.

Function templates

A function template is a prescription for the compiler to generate particular instances of a function varying by type. The emphasis here is on the compiler's role: this happens at compile time. The process of generating these instances is called *template instantiation* (or *monomorphisation* in other languages).

```
// T is a template type parameter
// the thing inside the <> is called a template parameter list
template<typename T>
auto min(T a, T b) -> T {
    return a < b ? a : b;
}

// because we are calling this templated function with T = int and T = double,
// the compiler will generate instances of the function min for these two types
// in the sense that there are two separate implementations present:
min(1, 2);     // auto min(int a, int b) -> int
min(0.9, 2.3); // auto min(double a, double b) -> double

// note the use of template argument deduction (?)
```

While this does slow down compilation and makes binaries larger, it is often advantageous to do this work upfront before runtime as it improves performance once the code is actually run, as there are less runtime checks incurred.

Type and non-type parameters

A template type parameter has unknown type and no value. A non-type parameter has known type with unknown value.

An example of how this might be useful is writing a generic procedure to find the minimum element of a `std::array`. These have fixed size which is specified as a template parameter, so we cannot simply parameterise the function over its element type.

```
template <typename T, std::size_t sz>
auto min_elem(std::array<T, sz> const a) -> T {
    // find the smallest element in a and return it
}

// compiler deduces what T and sz should be from a
```

Class templates

We can do similar things for classes as well, creating *class templates*:

```
template <typename T>
class X {
    T foo_;

public:
    X(T foo) : foo_{foo} {}

    auto get_foo() -> T {
        return foo_;
    }

    auto set_foo(T foo) -> void {
    }
};

// use like this
auto x1 = X<int>{1};
auto x2 = X<std::string>{"hi"};
```

Inclusion compilation model

Templated functions/classes *must* be defined in header files, because template definitions have to be known at compile time. This is in contrast to the usual link-time instantiation we use when writing non-polymorphic code that can separate the interface and implementation freely.

This can cause problems though, since it technically exposes implementation details in the interface, but also because it might make compilation a bit slower.

If in the above example the `set_foo` member function was never used, then no code is actually generated for that member function. This is called *lazy instantiation*. The same is not true for non-templated classes (i.e. if a class is non-templated and has member functions which are technically not used by anything, they are still generated anyhow).

Static members and friends of templated classes

Each template instantiation of a class has its own set of static members, as well as friend functions.

```
template <typename T>
class X {
    T foo_;

    // each X<T>, X<U>, X<V> instantiation has its own bar_ member
    static int bar_;

public:
    X(T foo) : foo_{foo} {}

    auto get_foo() -> T {
        return foo_;
    }

    auto set_foo(T foo) -> void {
    }

    // each X<T>, X<U>, X<V> instantiation has its own operator<< friend
    friend auto operator<<(std::ostream& os, X const& x) -> std::ostream& {
        // ...
    }
};
```

Metaprogramming and advanced templates

Constant expressions

A *constant expression* is a variable that can be calculated at compile time (as `#define`'d values are in C), or a function that, if its inputs are known, can be run at compile time (and its result substituted in place of that function call). We use the `constexpr` keyword to denote such things:

```
constexpr int fact_ce(int n) {
    return n <= 1 ? 1 : n * fact_ce(n - 1);
}

int fact(int n) {
    return n <= 1 ? 1 : n * fact(n - 1);
}

// can easily be calculated at compile time
constexpr int n = 10 + 20;

// function call is evaluated at compile time
// the omission of the constexpr keyword here means that the compiler is allowed
// to turn this into a constant expression if it wants to, but doesn't have to
// OTOH if we did specify it as a constexpr int n_fact_ce, the compiler must do it
int n_fact_ce = fact_ce(10);

// not evaluated at compile time, because fact isn't marked constexpr
int n_fact = fact(10);
```

This has two benefits, where applicable:

- We are offloading runtime computation to compile time computation, so get faster programs
- Potential errors can be flagged at compile time rather than at runtime, making them easier to pick up on

However, the natural downside is that this workload slows compilation.

Default members

We can set defaults for template type parameters:

```
// if cont_t is not actually specified when invoking an instance of this templated class,
// then its default value will be std::vector<T>
template <typename T, typename cont_t = std::vector<T>>
class stack {
public:
    // interface here ...
private:
    cont_t stack_;
};
```

Now when instantiating `stack`, one can give just the element type and fall back on a container type of `std::vector` if that fits the use case.

An example of this being done in practice is `std::vector`, which can take a second type parameter for a custom element allocator. Since it is uncommon to want to do this, it has a sane default template type parameter set in this case.

All template parameter lists (e.g. in member function definitions placed outside of the class declaration) have to be updated to conform as well if a template parameter is given a default value:

```
template <typename T, typename U = int>
class X {
public:
    auto f() -> T;
    auto g() -> T;
};

template <typename T, typename U>
auto X::f() -> T {
    // ...
}

template <typename T, typename U>
auto X::g() -> T {
    // ...
}
```

Template type parameters with defaults have to be placed at the end of the template parameter list, so it is not valid to start a template like

```
template <typename X, typename Y = int, typename Z>
```

Specialisation

If we want to give a more specific implementation of a templated type, we can use *specialisation* in two ways:

- *Partial specialisation* for a template for a type "based on" a template type parameter (e.g. a specialisation of a template for `T*` or `std::vector<T>`)
- *Explicit specialisation* for a fully-realised type (e.g. `std::string`, `int`)

Specialising a template is a good idea if:

- You need to preserve the existing semantics of a template for something that wouldn't otherwise work with the default generic implementation
  - Specialising to give completely different semantics/break assumptions about the behaviour of a class to other realised type parameters is poor form
- You're writing a type trait
- There is an optimisation to be had with specialisation (e.g. `std::vector<bool>` is fully specialised to improve space efficiency)

It is a bad idea to specialise functions, because they cannot be partially specialised, and explicit specialisation is better done via overloading. For this reason, explicit specialisation is only to be done on templated classes.

```
// partial specialisation
// note that we've given a partial amount of info about what the template type is,
// namely that it's a pointer, hence the qualifier "partial"
template <typename T>
class stack<T*> {
public:
    // some interface here

    auto sum() -> int {
        // here instead of summing by value naively, we would probably write
        // an implementation that dereferenced each value in the std::vector
        // this would make much more sense than summing by address (!!!)
    }

private:
    std::vector<T*> stack_;
};

// explicit specialisation
// note the use of an empty template parameter list
template <>
class vector<bool> {
public:
    // regular old interface

private:
    // here instead of storing a bool[], we might use some other space-efficient
    // representation, since bools occupy only 1 bit of memory instead of the 8
    // which are packed into a byte
}
```

(Aside: this `std::vector<bool>` specialisation exists in C++, but has come to be seen as a bit of a mistake in retrospect.)

Type traits

Type traits are a mechanism to introspect about the properties of types in C++, which can be helpful when you're working with templated types. These traits either allow you to ask questions about the type or make transformations to types (e.g. adding/removing `const`).

Traits that ask questions about types include things like

- `std::numeric_limits<T>`, which provides the minimum and maximum values of a type
  - This is in contrast to the C way to do this via `#define`s
- `std::is_signed<T>`, which provides a way to tell whether a type is signed (e.g. signed integer) or not

The "answer" to the question will be in some field of the trait (and the traits themselves usually take the form of a `struct`).

"Question traits" can be used to do conditional compilation, in combination with constant expressions:

```
auto algorithm_signed(int i) -> void;
auto algorithm_unsigned(unsigned u) -> void;

template <typename T>
auto algorithm(T t) -> void {
    // provided that the conditional expression is a bool constant expression itself,
    // if constexpr can resolve which conditional branch to use at compile time
    // in this case, we keep the algorithm for the appropriate signedness of T, and
    // throw a static (compile time!) error if this isn't possible
    if constexpr(std::is_signed<T>::value) {
        algorithm_signed(t);
    }
    else if constexpr(std::is_unsigned<T>::value) {
        algorithm_unsigned(t);
    }
    else {
        static_assert(std::is_signed<T>::value || std::is_unsigned<T>::value, "must be signed or unsigned");
    }
}
```

Traits used for type transformations include things like `std::move`, which under the hood uses a type trait (called `std::remove_reference`) to perform a conversion to an rvalue reference.

Variadic templates

Template parameter lists can be of a variable length:

```
// this acts as a "base case"
template<typename T>
T sum(T v) {
    return v;
}

// this acts as a "recursive case"
// typename... Ts is called a template parameter pack
// Ts... vs is called a function parameter pack
template<typename T, typename... Ts>
T sum(T v, Ts... vs) {
    // vs contains the parameter list less one value, so we are doing some proper
    // recursion here, although keep in mind this is all happening at compile time
    return v + sum(vs...);
}
```

We can go quite far with this and replicate pattern matching if we liked:

```
template<typename T>
bool pairwise_cmp(T v1) {
  return false;
}

template<typename T>
bool pairwise_cmp(T v1, T v2) {
    return v1 == v2;
}

// here, we can actually "capture" the first 2 values instead of just 1,
// and then do variadic template "recursion" as per usual
// because we might be given an odd number of arguments though, we either
// get a compile time warning if no single-arg base case templated function
// is given, or are forced to implement one with a sensible behaviour
// (perhaps always returning false if we're pairwise comparing an odd #. of elements)
template<typename T, typename... Ts>
bool pairwise_sum(T v1, T v2, Ts... vs) {
    return v1 == v2 && pairwise_cmp(vs...);
}
```

This can also be extended to variadic templated classes: see tuple as an example.

```
// must wrap chain in a struct to allow partial template specialization
template <int i, class F>
struct multi {
    static F chain(F f) {
        return f * multi<i - 1, F>::chain(f);
    }
};

template <class F>
struct multi<2, F> {
    static F chain(F f) {
        return f * f;
    }
};

template <int i, class F>
F compose(F f) {
    return multi<i, F>::chain(f);
}

// this prints out 10
auto increment = std::bind(std::plus<>(), std::placeholders::_1, 1);
std::cout << compose<10>(increment)(0) << "\n";
```

Member templates

If we wanted to support conversion between one templated class to another templated class (i.e. conversion of a stack of `int`s to a stack of `double`s), then we can use member templates to achieve this:

```
template <typename T>
class stack {
public:
    // this is a member function that is itself templated by some other type
    template <typename U>
    stack(stack<U>&);

    // other stuff here

private:
    std::vector<T> stack_;
};

// when giving the definition of the class like this, the extra template type must be
// treated as an "inner" templated type
// so this would not be the same as template <typename T, typename U>
template <typename T>
template <typename U>
stack<T>::stack(stack<U>& s) {
    while (!s.empty()) {
        stack_.push_back(static_cast<T>(s.pop()));
    }
}
```

Template template parameters

Template parameters may themselves be templates (e.g. `std::vector`) as opposed to fully-realised types.

```
// be very careful when specifying the template parameter list of template template parameters
// if we were trying to use std::vector here, it technically takes in two template params
// but we can use variadic template template parameters (!!!) to fix this
// we at least want one template arg for cont_t though (the type)
template <typename T, template <typename, typename...> typename cont_t>
class stack {
    // ...
private:
    cont_t<T> container_;
};
```

This allows us to write things like

```
// make it implicit that the vector has ints
auto s = stack<int, std::vector>{};
```

instead of

```
// must explicitly state that the vector has ints - blergh
auto s = stack<int, std::vector<int>>{};
```

Extension: SFINAE

TODO: substitution failure is not an error

Template argument deduction

The process by which the compiler determines what the types of type parameters and values of non-type parameters should be from the function arguments being used with them.

```
template <typename T, std::size_t sz>
auto min_elem(std::array<T, sz> a) -> T {
    T min = a[0];
    for (auto i = 0; i < sz; ++i) {
        min = min < a[i] ? min : a[i];
    }
    return min;
}

// deduces that T = int, sz = 4
auto min = min_elem(std::array<int, 4>{1, 2, 3, 4});
```

This works for variadic templates too.

Template argument deduction means that it is technically not necessary to write things like

```
std::vector<int>{1, 2, 3};
```

when the compiler could quite easily deduce that in

```
std::vector{1, 2, 3};
```

each element is of type `int`. This is called *implicit deduction.* But specifying the type (as in *explicit deduction*) can be used to leave the compiler in no doubt as to what the template values should be.

Extension: CRTP

TODO: the curiously recurring template pattern

Advanced types

`decltype`

Like many other languages, C++ allows you to determine at compile time what the type of an expression is. Unlike other languages (e.g. Python), this is purely a type-level mechanism, and does not allow for things such as

```
if type(a) == int:
    # do something
```

Rather, it is intended to be used for things such as

```
auto compare = [] (my_class a, my_class b) {
    // ...
};
std::priority_queue<my_class, std::vector<my_class>, decltype(compare)> pq(compare);
```

There are some rules for how this works, given an occurrence of `decltype(e)`:

1. If `e` is a variable in local or namespace scope, a static member variable or a function parameter, then the result is the variable/parameter's type `T`
2. If `e` is an lvalue (e.g. a reference), then the result is `T&`
3. If `e` is an xvalue (e.g. an rvalue reference returned by `std::move()`), then the result is `T&&`
4. If `e` is a prvalue (e.g. an integer literal), then the result is `T`

This mechanism can be useful for determining return types of templated code:

```
template <typename T, typename U>
auto add(T& lhs, U& rhs) -> decltype(lhs + rhs) {
    return lhs + rhs;
}
```

A trailing return type is crucial here, since

```
template <typename lhsT, typename rhsT>
decltype(lhs + rhs) add(lhsT& lhs, rhsT& rhs) {
    return lhs + rhs;
}
```

would not compile (as `lhs` and `rhs` have been used before they are declared).

Binding

This relates to what kind of references may be bound to what kind of function arguments:

Value type / Does it bind to this argument type?	`T&`	`T const&`	`T&&`	`template T&&`
lvalue	Yes	Yes		Yes
`const` lvalue		Yes		Yes
rvalue		Yes	Yes	Yes
`const` rvalue		Yes		Yes

Everything binds to `const` lvalue references (since we are in essence requesting a read-only view of some value). This includes rvalues too (which are "immutable")! Similarly, everything binds to templated rvalue references.

Non-`const` lvalue and rvalues only bind to references of that same type (i.e. a non-`const` lvalue only binds to a non-`const` lvalue reference), because it is a compile error to drop the `const` qualifier of a type.

Forwarding references (or universal references)

If a variable or parameter is declared to have type `T&&` for some deduced type `T`, that variable or parameter is called a *forwarding reference*. The requirement that type deduction is involved is critical, because everything binds to universal references (i.e. the `template T&&` from before). If there is a non-directly deduced type, then it is instead an rvalue reference.

```
int n;

// lvalue reference
int& lvalue = n;

// no deduced parameter type => rvalue reference
int&& rvalue = std::move(n);

// deduced parameter type => universal reference
template <typename T> T&& universal = n;

// also a universal reference!
auto&& universal_auto = n;

// param is of universal reference type
template<typename T>
void f(T&& param);

// param1 and param2 are rvalue references, because type deduction has to be
// directly involved here
// (so saying it has to be `T&&` for deduced type `T` is strict!)
template<typename T>
void f(std::vector<T>&& param1, T const&& param2);
```

Reference collapsing

There are rules around determining what types like `T& &&`, called the *reference collapsing rules*:

- rvalue references to rvalue references become rvalue references (i.e. `T&& &&` -> `T &&`)
- All other references of references collapse to lvalue references
    - `T& &` -> `T &` (lvalue ref of lvalue ref)
    - `T&& &` -> `T &` (lvalue ref of rvalue ref)
    - `T& &&` -> `T &` (rvalue ref of lvalue ref)

Forwarding functions

When writing a templated wrapper function, some problems arise:

```
// fails to compile if T is non-copyable (i.e. its copy ctor/assn is deleted)
// also not very good if values of type T are expensive to copy
template <typename T>
auto wrapper(T v) -> auto {
    return fn(v);
}

// not very useful if fn needs to modify v
// also doesn't behave as intended if we pass an rvalue via std::move
// since wrapper(std::move(v)) will call fn(v) instead of fn(std::move(v))
template <typename T>
auto wrapper(T const& v) -> auto {
    return fn(v);
}

// fails if we pass it a const lvalue reference
// fails to compile if given an rvalue
template <typename T>
auto wrapper(T& v) -> auto {
    return fn(v);
}

// doesn't fail to compile anymore, but v is not actually an rvalue inside
// the function, but rather an lvalue (so fn is passed an lvalue)
// this is because named rvalue references are lvalues
template <typename T>
auto wrapper(T&& v) -> auto {
    return fn(v);
}
```

So that lvalues are treated as lvalues and rvalues are treated as rvalues in this context, we could `static_cast` on `v` before calling it, but there is a more readable alternative in `std::forward`:

```
template <typename T>
auto wrapper(T&& v) -> auto {
    return fn(std::forward<T>(v)); // =~ return fn(static_cast<T&&>(v));
}
```

Prime examples of where this might be useful is in something like `make_unique`, which takes in arguments for constructing a `unique_ptr<T>` value:

```
template <typename T, typename... Ts>
auto make_unique(Ts&&... args) -> std::unique_ptr<T> {
    // note that the ... is outside the forward call, and not right next to args,
    // because we want to call
    // new T(forward(arg1), forward(arg2), ...)
    // and not
    // new T(forward(arg1, arg2, ...))
    return std::unique_ptr(new T(std::forward<Ts>(args)...));
}
```

By doing this, value categories are preserved even though the arguments to `make_unique` may be a mix of lvalues and rvalues. This is called *perfect forwarding*.

You should use `std::forward` when you want to wrap functions with a parameterised type, which you might want to do for a few reasons (e.g. doing something special before/after the function call).

Testing:

Explicit instantiation of templates

`template class my_template<A, B>;` to explicitly instantiate `my_template` with type parameters as `A` and `B`, which instantiates any non-templated member functions as well (see if it compiles).

Do similar with functions as:

```
template <typename T> void func(T param) {} // definition
template void func<int>(int param); // explicit instantiation.
```

Templated test type for parameters is useful, make sure all assumable operations / functions are templated to ensure that your own templates don't rely on implicit conversion in the definition, since those aren't considered with template type deduction:

```
template<typename T>
struct template_tester {
    int value;
};

template<typename T>
auto operator<<(std::ostream& os, const template_tester<T>& rhs) -> std::ostream& {
    os << rhs.value;
    return os;
}

template<typename T>
auto operator<=>(const template_tester<T>& lhs, const template_tester<T>& rhs) -> std::strong_ordering {
    return lhs.value <=> rhs.value;
}

template<typename T>
auto operator==(const template_tester<T>& lhs, const template_tester<T>& rhs) -> bool {
    return lhs.value == rhs.value;
}

template<typename T>
struct std::hash<template_tester<T>> {
    std::size_t operator()(const template_tester<T>& t) const {
        return std::hash(t.value);
    }
};
```

Example catch2 Tests:

```
TEST_CASE("Show how Catch2 works.") {
    std::cout << "this runs before every SECTION";

    SECTION("Require") {
        REQUIRE(0 == 0);
    }

    SECTION("Check") {
        CHECK(0 == 0);
        CHECK_FALSE(0 == 1);
    }

    SECTION("Exceptions") {
        // check that it throws, don't terminate program
        CHECK_THROWS(foo());
        // check that it doesn't throw, if it does then don't terminate the program
        CHECK_NOTHROW(bar());

        // check that it throws this specific type
        CHECK_THROWS_AS(foo(), std::domain_error);
        // check that it throws this specific message
        CHECK_THROWS_WITH(foo(), "assert this as message");

        // check that it throws a std::runtime_error
        // with the message "assert this as message"
        CHECK_THROWS_MATCHES(foo(), std::runtime_error,
                             Catch::Matchers::Message("assert this as message"));
    }
}
```