

COMP6771

Advanced C++ Programming

1.1 Course Outline



UNSW
SYDNEY

Teaching Staff

Lecturer in Charge

Imran Razzak

Course Admin

Simon Haddad

Potential Guest Lecturers

Optiver

Nathaniel Shead

Tutors

Jack Peng

Jayden Leung

Junyuan Zhou

Peiyu Tang

Rory Golledge

Simon Haddad

Sunny Chen

Course Objectives

You will develop:

1. Skills in writing software using modern C++.
2. Skills in writing unit tests to create robust software.
3. Skills in using libraries to develop software.
4. Skills in using tools to build software.
5. Knowledge & understanding about imperative, generic, and object-oriented programming.

What is C++?

- General-purpose programming language evolved from C.
- Created by Bjarne Stroustrup in 1979.
- Backwards-compatible with C.
- Designed to run natively, directly on hardware.
- Only language lower than C++ is assembly.
- Supports development of **zero-overhead & opt-in overhead abstractions**.
- Multi-paradigm: imperative, generic, object-oriented, functional.

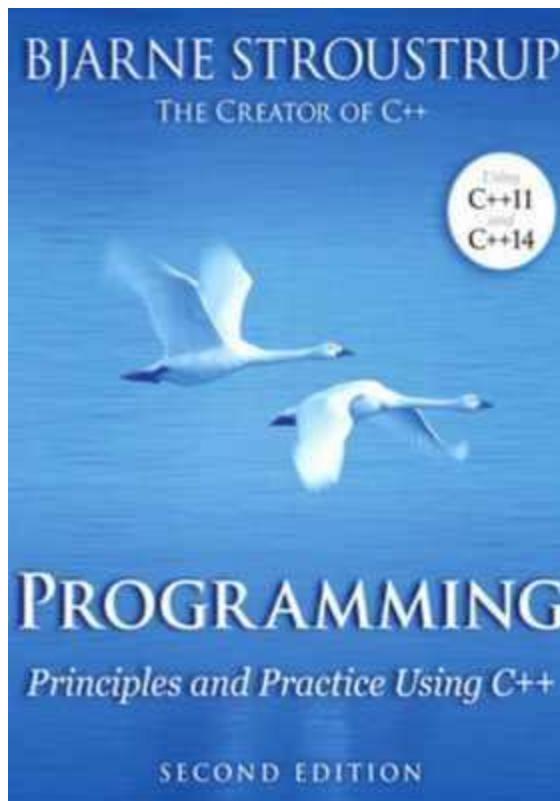
What isn't C++?

- **C++ is not C!**
- Easy to think you can build your C++ understanding on top of your C understanding.
- Valid C code is often (but not always!) valid C++ code, but good C is very different from good C++.
- C and C++ have a large common intersection but are distinct languages.
For example, in teaching best practices, we will not be using:
 - `malloc()`/`free()`
 - C-style arrays
 - C-style strings
- And will sometimes discourage the use of raw pointers (`int*`, `char*`, etc.)

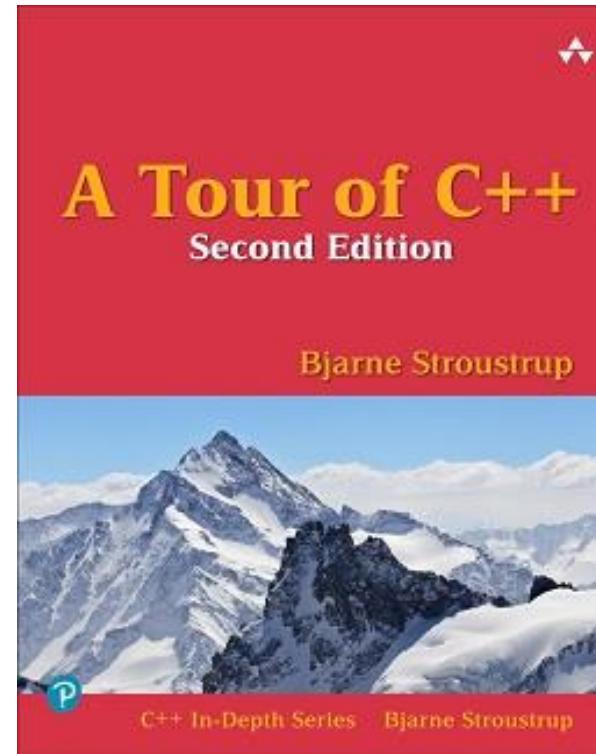
What is C++ Good For?

- Operating systems
- Low latency software (e.g., high frequency trading)
- Direct control of hardware
- Game Development
- Implementations of *other* programming languages (e.g., NodeJS)
- Just about anything you can think of!

Learning Resources



Comprehensive intro to C++
(>100 pages of exercises!)



Will help you in a pinch (e.g. before exams and interviews)
Readable in 4 hours.

cppreference

- Good for looking up APIs and recalling language rules
- **DO NOT USE CPLUSPLUS.COM**

Where to Get Help

- Your question/answer hierarchy:
- Edstem forum.
- Your tutor (see Timetable page for links)
- Lecturer (cs6771@cse.unsw.edu.au)
- Imran (imran.razzak@unsw.edu.au)

Only questions that are non-sensitive will be answered on the forum.

Course Schedule

See the [course outline](#) for the full schedule.

Weekly teaching includes:

- 4 hours of lectures
- 2 hours of labs
- Incremental development on assignments

We may provide additional material and webinars to assist in your learning. While these will be recommended, they will not be required.

Assessment Schedule

Assessment	Weighting	Due Date
Weekly Exercises	10%	Weeks 1 – 5, 7 – 10 (Sunday of that week, 8pm)
Assignment 1	15%	Late Week 3 (see assignment spec)
Assignment 2	20%	Early Week 7 (see assignment spec)
Assignment 3	25%	Early Week 10 (see assignment spec)
Exam	30%	Exam Period

Exam Assessment

Final exam **may** be scaled.

Final exam:

- No hurdle

Assignments:

- have an emphasis on correctness
- rely on version control (assumed knowledge)
- have a late penalty outlined in the specification

Plagiarism will not be tolerated.

- Immediate zero for assignment.

Course Tools

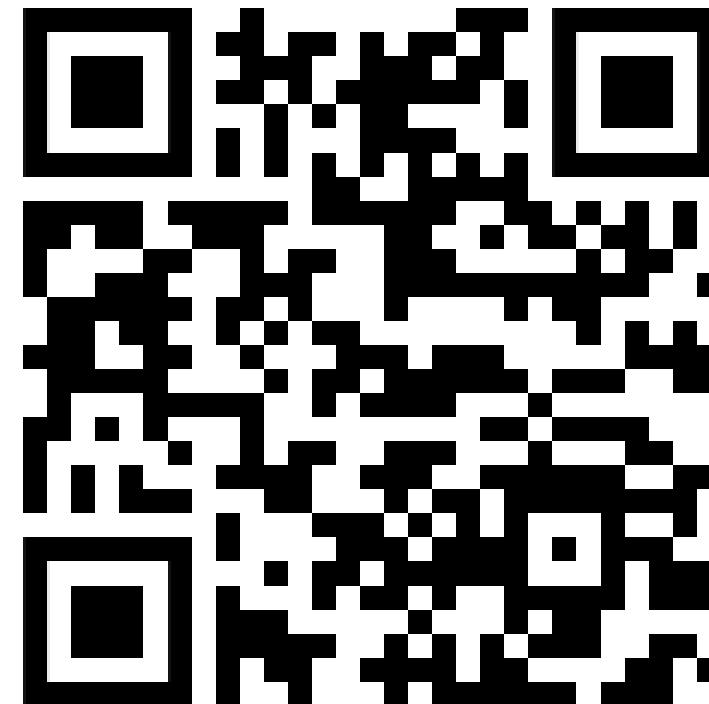
Course Website on [WebCMS3](#)

Tutorials & Assignments distributed through [Gitlab](#)

Developer Environment:

- Build C++ using [CMake](#)
- C++ Compiler (must have C++20 support): GCC 10+, Clang 12+
- Recommended Editors: VS Code, CLion
- Documentation: [cppreference](#)
- Git

Feedback (stop recording)



COMP6771

Advanced C++ Programming

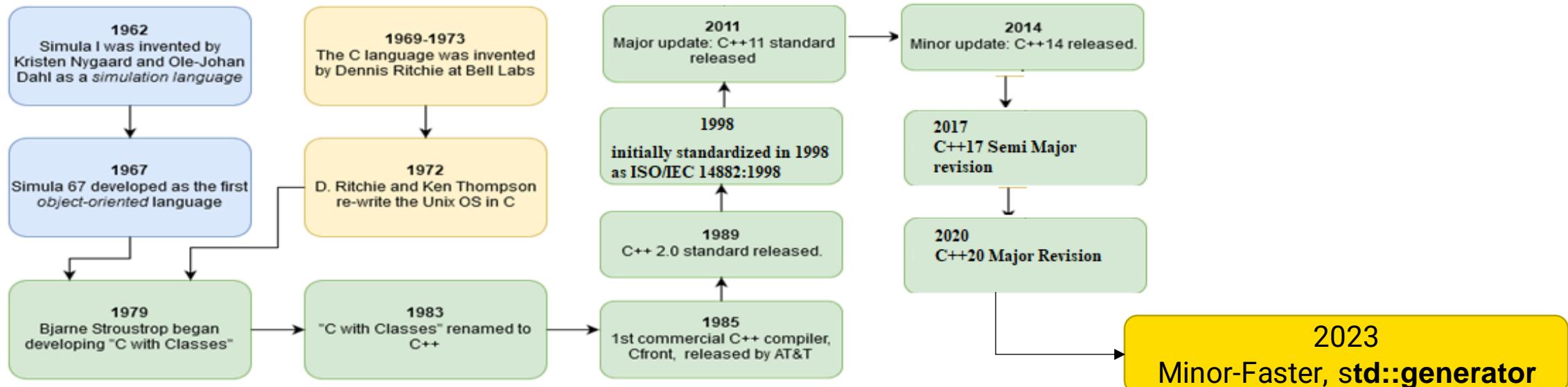
1.2 C++ Fundamentals



UNSW
SYDNEY

C++ Standards

- C++ is an International Standards Organisation (ISO) language.
- Original standard was released in 1998, known as C++98.
- Since 2011, a new revision of the standard has been released every 3 years.

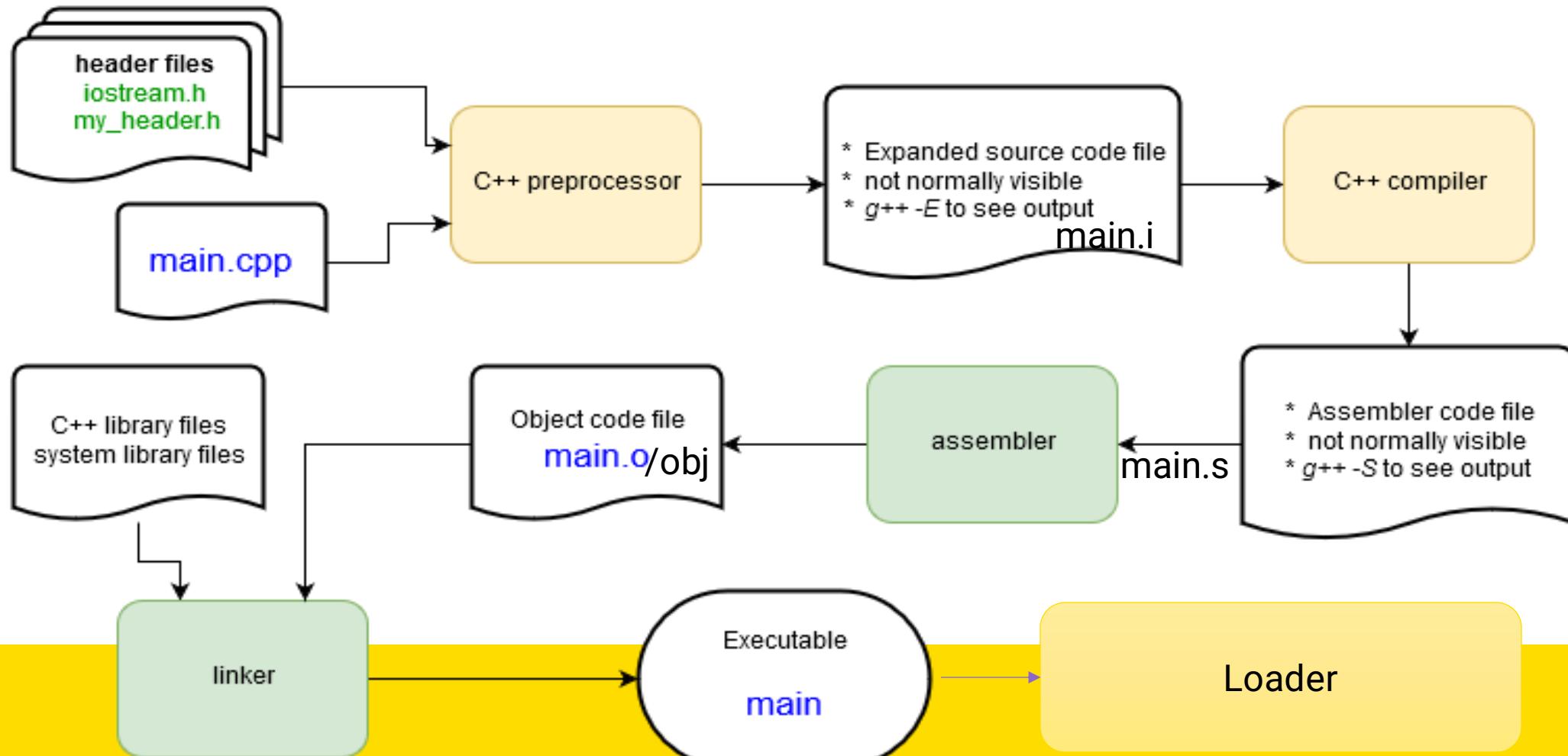


- This course teaches modern C++20.
- On older compilers, some topics and features may not be available.

Behind the Scene

```
#include <iostream>

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



loads a *header* file containing function and class definitions

```
#include <iostream>

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

The **return** statement returns an integer value to the operating system after completion. 0 means “no error”. C++ programs **must** return an integer value.

namespace called *std*. Namespaces are used to separate sections of code for programmer convenience. To save typing we'll always use this line in this tutorial.

The *main* routine – the start of **every** C++ program! It returns an integer value to the operating system and (in this case) takes no arguments: *main()*

- C++ (along with C) uses *header files* as to hold definitions for the compiler to use while compiling.
- A source file (file.cpp) contains the code that is compiled into an object file (file.o).
- The header (file.h) is used to tell the compiler what to expect when it assembles the program in the linking stage from the object files.
- Source files and header files can refer to any number of other header files.

Fundamental Types

A **type** in C++ is made up of:

- Certain storage requirements (e.g., 4-bytes, 8-bytes, etc.)
- A set of allowable values (e.g., -1 for int)
- A set of allowable operations (e.g., addition)

C++ has many of the same fundamental types as C, but there are a few more

```
// From C
int meaning_of_life = 42;
unsigned u = 6771u;
double six_ft = 1.8288;
char letter = 'c';
const char *str = "COMP6771!";
float pi = 3.14f;
void(*my_free)(void *ptr) = free;
// NEW! In C++, the Boolean type
bool t_or_f = false;
```

Fundamental Types & Portability

Remember:

- C++ runs directly on hardware.
- Fundamental type sizes may change depending on what machine your code is run on
- E.g., long being 32- vs. 64-bit.

Use the Standard Library to help inspect the system at runtime.

```
#include <iostream>
#include <limits>

int main() {
    int imax = std::numeric_limits<int>::max();
    int smin = std::numeric_limits<short>::min();

    std::cout << imax << std::endl;
    std::cout << smin << std::endl;
}
```

User-Defined Types (UDT)

- Users can create their own types through combining fundamental types and **structs**, **classes**, and **unions**.
- Many built-in objects in higher-level languages are implemented in C++ as UDTs.
- The C++ Standard Library provides many ready-to-go types.

```
// a bona-fide string class  
std::string text = "process me!";  
  
// dynamic array of integers  
std::vector<int> ints = {1, 2, 3};  
  
// an associative map  
std::map<int, int> dict = {{3, 1}};  
  
// function object (functor)  
std::function<void(void*)> my_free = free;
```

Enumerations & Enum Classes

C++ supports enumerations from C.

Enumerations are named symbolic constants implemented as some integral type.

New in C++: enumeration classes:

- C-style enumerations are freely usable as integers, which could lead to bugs.
- C++ enum classes **cannot** be used as integers.
- The underlying integer type can be selected.

Enum classes are the preferred way of making symbolic constants

```
enum color { RED = 0, GREEN, BLUE, };  
enum class rgb : unsigned char {  
    R = 0,  
    G,  
    B,  
};  
assert(color::RED == rgb::R); // true  
  
// ERROR: enum class members do not support  
// bitwise-OR by default (unlike ints)  
rgb::R | 0x2;
```

const

- The `const` type modifier makes a value **immutable**.
- Idiom of **const-correctness**:
 - Everything should be `const`
 - ...unless it needs to be modified.
- We will focus on const-correctness as a major topic.
- `const` can appear to the left or the right of a type:
 - “east”-`const` vs. “west” `const`
 - You can use either, just be consistent

```
// west const
const int ci = 42;
// east const
int const ic = 6771;
// the below will not compile.
ci++;

int i = ci;
// OK because we copied ci into i
i++;
```

Top-level & Bottom-level const

- Pointers innately have two associated pieces of data:
 - The pointer.
 - The pointee (what's being pointed to).
- Top-level const:
 - The *pointer* is constant and cannot point to anything else.
- Bottom-level const:
 - The *pointee* is constant and cannot be modified through this pointer.
- A variable can be both top-level and bottom-level at the same time.



```
int i = 0;  
// top-level const  
int * const p = &i;
```



```
int i = 0;  
const int *p = &i;  
// bottom-level const only
```



constexpr

- `constexpr` is much like `const` except that the value *must* be known or calculable at **compile-time**.
- `constexpr` variables replace `#define` macros from C
- Unlike macros, `constexpr` variables are affected by scope and are type-checked.
- A `const` object initialised from a `constexpr` is NOT also a `constexpr`.

```
constexpr int N = 4;
int get_int(); // defined elsewhere

int main() {
    const int M = get_int();
    // not OK: M not known until runtime
    int arr[M] = {0};

    // OK: N is a constexpr variable
    int arr[N] = {0};

    int a=0;
    std::cin>>a;
    int const b=a+1; // OK Can be computed at runtime
}
```

Why `const` or `constexpr`

- Clearer code (you can know a function won't try and modify something just by reading the signature).
- Immutable objects are easier to reason about.
- The compiler *may* be able to make certain optimisations.
- Immutable objects are much easier to use in multithreaded situations.

Expressions

- In Computer Science, an expression is a sequence of values and operations that are combined to produce a new value.
- C++ supports the same operators as C.
- It also provides a few new operators.
 - and, or, and not are synonyms for &&, ||, and !

```
// some arithmetic expressions
int i = 3, j = 4;
double k = (1 + i) * 3 - j / 0.5;
// some boolean expressions
bool am_hungry = true;
bool is_dinner_time = false;
bool on_a_diet = true;
bool will_eat =
    (dinner_time or am_hungry) and not
on_a_diet;
// will I eat?
```

Type Conversions

- C++ has **implicit** and **explicit** types conversions
- For fundamental types, the same rules as in C are followed.
- For User-Defined Types, we will cover this later in the course.

```
// Implicit promoting conversion
int i = 42;
double d = 1.5;
float promoted = i * d;
// i is promoted to a float
// then the product is converted to a
float
```



```
// Explicit narrowing conversion
double d2 = 42.5;
int narrowed = static_cast<int>(d2);
```

C++ Has Value Semantics

```
std::string s1 = "C++ has no implicit references like Java";
```

```
// s1 is copied into s2.
```

```
std::string s2 = s1;
```

```
// though they are equal...
```

```
assert(s1 == s2);
```

```
// they do not share the same memory!
```

```
assert(s1.data() != s2.data());
```

C++ References

C++ supports C-style pointers, but also offers **references**.

A reference is an alias to another object; it can be used as you would the original.

A reference:

- Has no need to use “->” to access members.
- Cannot be null (no null references).
- Once bound to an object cannot be rebound.

Under the hood, references are implemented as pointers.

```
float pi = 3.14f;  
float &pi_ref = pi;  
  
pi_ref = 3.5;  
  
// true: pi_ref is just an alias  
// for pi  
assert(pi == 3.5);
```

References & const

- A reference to `const` means you cannot modify the original object through *this* reference.
- It may still be possible to modify the original object through another reference.
- Note that the references are always top-level `const`, but can optionally be bottom-level `const`.

```
int i = 1;
const int &ref = i;
std::cout << ref << '\n';
i++; // This is fine
std::cout << ref << '\n';
ref++; // This is not

const int j = 1;
const int &jref = j; // this is allowed
int &ref = j; // not allowed
```

Type Inference with auto

- Use `auto` to let the compiler statically infer the type of a variable based on what is being assigned to it!
 - **Almost Always Auto (AAA):**
 - A style philosophy that says to put `auto` everywhere it can go
 - We do not follow AAA, but if you use `auto` you should use it consistently
- ```
auto i = 0; // is an int
auto d = 0.0; // d is a double.
auto u = 0; // is u unsigned? No!
auto uu = 0u; // now uu is unsigned.
auto b = i == 0; // b is a Boolean

auto c = 'c'; // c is a char
auto str = "comp6771"; // what is str?
// if you guessed const char *,
// you are correct!
```

# Statements: if

- C++ supports the classic if-statement from C.
- It also supports the ternary operator from C as well.
- Sometimes, using the ternary operator can simplify variable initialisation and make code simpler.

```
char c = get_char();
int i;
if (c == 'd') {
 i = 42;
} else {
 i = 43;
};

// could also be written
// as...
int i = c == 'd' ? 42 : 43;
```

# Statements: switch

C++ supports the switch-statement from C.

New in C++:

- The `[[fallthrough]]` attribute can be used to signify you intended for a case to fallthrough.
- Improves the readability of code and can *sometimes* enable optimisation.

```
auto b = get_bool();
switch(b) {
 case true: [[fallthrough]]
 case false: [[fallthrough]]
 default:
 std::cout << b << std::endl;
}
```

# Statements: while, do-while, for

C++ supports the same loops as C

New in C++:

- The ranged-for loop simplifies looping over whole sequences.
- “Element-based” iteration vs. “index-based” iteration.
- Most Standard Library containers also support ranged-for.
- Later, you will learn how to make your own types support ranged-for as well.

```
int iarr[4] = {1, 4, 9, 16};
for (int i = 0; i < 4; ++i) {
 std::cout << i; <<
 std::endl;
}

// Could also be written as...
for (int i : iarr) {
 std::cout << i <<
 std::endl;
}
// the above works because the
compiler knows how big iarr is.
```

# Functions: Overview

- C++ supports functions just as in C.
- With auto, new function syntax
  - You can use either, just be consistent
- Functions still support pass-by-value from C.
- Functions now also support true pass-by-reference with references.
- C++ also supports default function parameters.
- Functions can be overloaded based on parameter types.

```
#include <iostream>

auto main() -> int { // auto style
 // print "Hello world" to the terminal
 std::cout << "Hello, world!\n";
}

int main() { // classic style
 // print "Hello world" to the terminal
 std::cout << "Hello, world!\n";
}
```

# Functions: Pass-by-Value

- The actual argument is copied into the memory being used to hold the formal parameter's value during the function call/execution
- All formal parameters are just local variables in the function.

```
#include <iostream>
void swap(int x, int y) {
 const int tmp = x;
 x = y;
 y = tmp;
}
int main() {
 int i = 1, j = 2;
 std::cout << i << ' ' << j << '\n'; // 1 2
 swap(i, j);
 // 1 2... No swap?
 std::cout << i << ' ' << j << '\n';
}
```

# Functions: Pass-by-Reference

- The formal parameter merely acts as an alias for the actual argument.
- Anytime the function uses the formal parameter (for reading or writing), it is actually using the original object.
- Pass-by-reference is useful when:
  - The argument cannot be copied.
  - The argument is large.

```
#include <iostream>
void swap(int &x, int &y) {
 int tmp = x;
 x = y;
 y = tmp;
}
int main() {
 int i = 1, j = 2;
 std::cout << i << ' ' << j << '\n'; // 1 2
 swap(i, j);
 std::cout << i << ' ' << j << '\n'; // 2 1
}
```

# Functions: Default Arguments

- Functions can use default arguments, which is used if an actual argument is not specified when a function is called.
- Default values are used for the *trailing* parameters of a function call - this means that ordering is important.
- Formal parameters: Those that appear in the function prototype.
- Arguments: Those that appear when calling the function.

```
int rgb(short r = 0, short g = 0, short b = 0);

rgb(); // same as rgb(0, 0, 0);
rgb(100); // same as rgb(100, 0, 0);
rgb(100, 200); // same as rgb(100, 200, 0)

rgb(100, , 200); // error
rgb(100, default, 200); // error
```

# Functions: Overloading

- Function overloading refers to a family of functions in the **same scope** that have the **same name but different formal parameters**.
- This can make code easier to write and understand.
- **Aim to write overloads that are trivial.**
- **If non-trivial to understand, name your functions differently.**
- It is possible to overload a function based on bottom-level const

```
int square(int x) {
 return x * x;
}

double square(double x) {
 return x * x;
}

square(3); // OK: int square(int) found
square(3.5); // OK: double square(double) found
square(3.5); // OK: float convertible to double
square(); // error: no square() function found
```

# Functions: Overload Resolution

- The function to call is determined by **overload resolution**:
    1. Find candidate functions (have the same name)
    2. Select viable ones (same number of arguments & each argument convertible)
    3. Find the best match (types much better in at least one argument).
    4. Return types are ignored in overload resolution.
  - Errors in function matching are found during compile-time.
  - Full details can be found [here](#)
- ```
auto g() -> void;
auto f(int) -> char;
auto f(int, int) -> void;
auto f(double, double = 3.14) -> short;

// g(): ignored (not called f)
// f(int, int): ignored (wrong number of args)
// f(int) vs. f(double, double)
// f(double, double) selected since no
// conversion needed to call, is better match
f(5.6);
```

Namespaces

- Namespaces are a way to prevent name collisions between different parts of code.
- Names inside a namespace are accessed with the scope operator ::
- We will discuss namespaces more in later in the course.

```
namespace nonstd {  
    char get_char();  
  
    int course = 6771;  
}  
  
// access via scope operator  
std::cout << nonstd::course << std::endl;  
auto c = nonstd::get_char();
```



Templates

- Templates are a way to write generic code in C++.
- We will discuss them in much more depth later in the course.
- Today we will briefly show their syntax

```
#include <vector>
#include <map>
// A vector of “int”. The type is specified in
// the <> angle brackets
std::vector<int> ints = {1, 2, 3};

// a mapping of int -> bool.
// the Key type is int
// the Value type is bool
std::map<int, bool> m = {{0, false}, {1,
true}};
```

Common Library Types

We will discuss the Standard Library more in Week 2.

Today we will discuss some of the most common types:

- `std::vector`, a dynamic array
- `std::set`, a hash set
- `std::map`, a hash map
- File I/O

Most of the standard library uses **templates** to provide generic code reuse.

Sequence Container: std::vector

- std::vector is an automatically growing dynamic array.
- Useful for almost any situation.
- Searching through a vector with a for-loop is extremely fast.

```
#include <vector>
#include <iostream>
std::vector<int> ints = {1, 2, 3};

assert(ints[2] == 3); // true
ints[0] = 4;

for (const int &i : ints) {
    std::cout << i << std::endl;
}
```

Hash Set: std::unordered_set

- std::unordered_set is a generic hash set.
- Can store and retrieve elements in constant time.
- As opposed to std::set, elements have no inherent ordering.

```
std::unordered_set years = {1996, 2006, 2020};  
  
assert(years.contains(1996)); // true  
  
years.insert(2016);  
assert(years.find(2016)); // true  
  
years.erase(2020);  
assert(!years.contains(2020)); // true
```

Hash Map: std::unordered_map

- std::unordered_map is a generic hash map.
- Retrieval of a key-value pair done in constant time.
- As opposed to std::map, keys are not stored in any inherent order.

```
std::map<int, char> ascii_dict = {  
    {32, ' '},  
    {0, '\0'}  
};  
  
assert(ascii_dict[32] == ' '); // true  
ascii_dict[65] = 'A';  
  
// many more operations
```

File I/O: std::ifstream, std::ofstream

```
#include <iostream>
#include <fstream>
int main () {
    auto fout = std::ofstream{"data.out"};
    // Below line only works C++17
    if (auto in = std::ifstream{"data.in"}; in) { // attempts to open file, checks it was opened
        for (auto i = 0; in >> i;) { // reads in
            std::cout << i << '\n';
            fout << i;
        }
        if (in.bad()) {
            std::cerr << "unrecoverable error (e.g. disk disconnected?)\n";
        } else if (not in.eof()) {
            std::cerr << "bad input: didn't read an int\n";
        } // closes file automatically <-- no need to close manually!
    }
    else {
        std::cerr << "unable to read data.in\n";
    }
}
```

Declarations & Definitions

A declaration makes known the type and the name of an entity.

A definition is a declaration, but also does extra things.:

- A variable definition allocates storage for, and constructs, a variable.
- A class/struct/union definition allows you to create variables of that type.
- You can call functions with only a declaration but must provide a definition later.

Everything must have precisely one definition after linking.

```
void declared_fn(int arg);  
class declared_type;  
  
// This class is defined, but not all the methods are.  
class defined_type {  
    int declared_member_fn(double);  
    int defined_member_fn(int arg) { return arg; }  
};  
  
// These are all defined.  
int defined_fn() { return 1; }  
int i; // at global scope, the default value is 0.  
const int j = 1;  
std::vector<double> vd = {};
```

Program Errors

- Four primary kinds of program errors:
 - Compile-time
 - Link-time
 - Run-time
 - Logic
- Errors are not the same as *exceptions*; they will be discussed later.

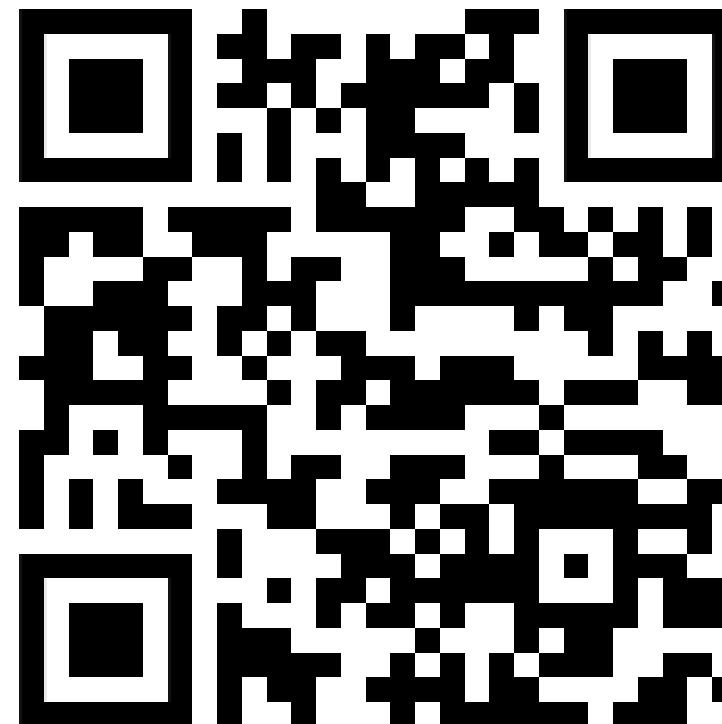
```
int main() {
    // compile-time error: no type given
    a = 5;

    // link-time error: no function definition
    char get_char();
    char c = get_char();

    // run-time error: file not found
    auto file = std::ifstream{"comp6771.txt"};

    // logic error: out-of-bounds memory
    int arr[4] = {0};
    arr[4] = 1;
}
```

Feedback (stop recording)



COMP6771

Advanced C++ Programming

2.1 – Standard Library Overview



UNSW
SYDNEY

Software Libraries

- Most of us are quite familiar with libraries in software. For example, when programming in C, we frequently use `<stdio.h>` and `<stdlib.h>`.
- Being an effective programmer often consists of the effective use of libraries. In some ways, this becomes more important than being a genius at writing code from scratch. Don't reinvent the wheel!
- It is essential to know what facilities your favourite language's Standard Library provides before using 3rd Party libraries!



C++ Standard Library

C++ offers a wealth of ready-to-use functions and types in its Standard Library:

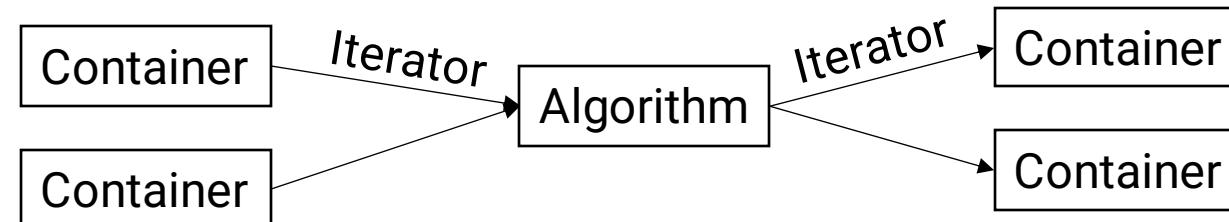
- The original C Standard Library
- A containers library
- An iterators library
- An algorithms library
- A new string library
- A memory management library
- A new random numbers library
- Ranges
- String
- And [many more!](#)

The containers, iterators, and algorithms libraries are collectively known
as the Standard Template Library.
We will be focusing on these.



STL: Standard Template Library

- STL is an architecture and design philosophy for managing generic and abstract collections of data with algorithms.
- All components of the STL are templates.
- Containers store data, but don't know about algorithms.
- Iterators are an API to access items within a container in a particular order, agnostic of the container used.
 - Each container has its own iterator types.
- Algorithms manipulate values referenced by iterators, but don't know about containers.



Why STL?

- STL offers an assortment of containers
- STL publicizes the time and storage complexity of its containers
- STL containers grow and shrink in size automatically
- STL provides built-in algorithms for processing containers
- STL provides iterators that make the containers and algorithms flexible and efficient.
- STL is extensible which means that users can add new containers and new algorithms such that:
 - STL algorithms can process STL containers as well as user defined containers
 - User defined algorithms can process STL containers as well user defined containers



Introductory Example

```
#include <algorithm>
#include <iostream>
#include <vector>

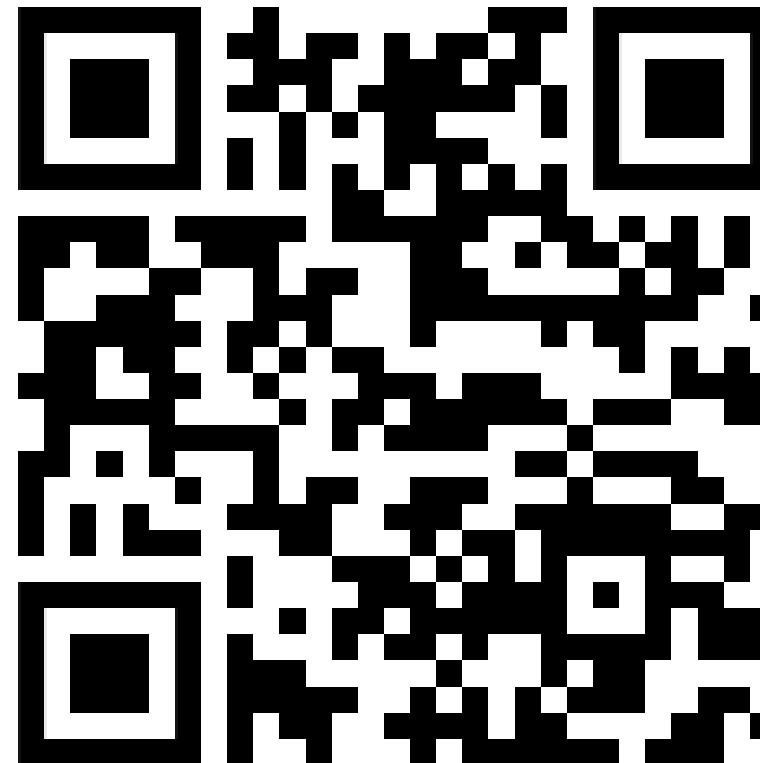
bool is_even(int n) { return n % 2 == 0; }

int main() {
    auto v1 = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8};
    auto v2 = std::vector(4); // makes space for 4 elems

    // use standard algorithm to copy only the elements that satisfy "is_even".
    // begin() and end() return iterators
    std::copy_if(v1.begin(), v1.end(), v2.begin(), is_even);

    for (int elem : v2) { // the ranged for-loop also secretly uses iterators!
        std::cout << elem << std::endl;
    } // prints 2, 4, 6, 8
}
```

Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

2.2 Standard Containers



UNSW
SYDNEY

Sequential Containers

Organises a finite set of objects into a strict linear arrangement.

std::vector	Dynamic array: Random Access, back insertion, C compatible, preferred choice
std::array	Fixed-size array
std::deque	Double-ended queue, Random access, Back and front insertion, Slower, no C compatible
std::list	Doubly-linked list, no random access, insert anywhere,
std::forward_list	Singly-linked list



Sequential: std::array

- Encapsulate fixed-size array
- Contiguous elements of the same type.
- Random access done in constant time.
- Doesn't decay to T^* when passed to functions
- Size must be known at compile time
- Passing to functions causes a copy
- `at()`, `[]`, `front()`, `back()`, `.data()`

```
#include <array>
#include <iostream>

auto arr = std::array<T, N>{};
auto arr = std::array<int, 3>{1, 2, 3};

std::cout << "1st element: " << arr.at(0) << "\n"; // slower due to bounds checking
std::cout << "2nd element: " << arr[1] << "\n"; // faster, less safe
std::cout << "3rd element : " << arr.back() << "\n";
for (int n : arr)
    std::cout << n << "\n";
```

Sequential: std::vector

- Dynamic array of contiguous elements.
- Grows on demand, shrinks on request.
- Random access is constant time.
- #include <vector> to use
- Best container to use in almost all cases:
 - Locality of elements allows vector to utilise modern hardware caches effectively
 - Dynamic array vs array list

```
#include <vector>
#include <iostream>

auto v = std::vector<int>{1, 2, 3};
std::cout << "1st element: " << v.at(0) << "\n"; // slower, safer
std::cout << "2nd element: " << v[1] << "\n"; // faster, less safe
std::cout << "Max size before realloc: " << v.capacity() << "\n";
for (int n : v)
    std::cout << n << "\n";
```

Associative Containers

- Almost all operations have logarithmic time complexity.
- Store only weakly strict ordered type (e.g., numeric types)
 - Must be comparable with the "<" operator.
- Sorting criteria customisable through the template parameters.
- Hashed versions available

<code>std::set</code>	Item stored act as key, no duplication
<code>std::multiset</code>	Duplication allowed
<code>std::map<K,V></code>	Separate key and value, no duplicate
<code>Std::multimap<K,V></code>	Map, allow duplication

Ordered Associative Containers

Provide fast retrieval of data based on ordered keys.

<code>std::set</code>	A collection of unique keys.
<code>std::map</code>	A collection of key/value pairs
<code>std::multiset</code>	A collection of keys
<code>std::multimap</code>	A collection of unique keys to many values

- * support insertion via copy and construction in-place of elements (`emplace`).
- * `emplace` is used to construct an object in-place to avoid unnecessary copy.
- * `emplace` and `insert` are equal for primitive data but for objects prefer `emplace()`

All ordered associative containers are implemented as linked data structures.



Associative: std::map

- Usually implemented as a red-black tree.
- Key-value pairs
- Logarithmic access time for most operations
- Keys default sorted in ascending order
- #include <map> to use
- Good as a fallback map type, but faster alternatives exist
 - Such as std::unordered_map

```
#include <map>

std::map<std::string, double> m;

m.insert({"bat", 14.75}); // The insert function takes in a key-value pair.
m.emplace("cat", 10.157); // This is the preferred way of using a map

// This has unintended side-effects and causes many bugs if misused.
std::cout << m["bat"] << '\n';
auto it = m.find("bat"); // Iterator to bat if present, otherwise m.end()

// Normally this is a place to use auto, but for demonstration purposes...
for (const std::pair<const std::string, double>& kv : m) {
    std::cout << kv.first << ' ' << kv.second << '\n';
}
```

Descending order:

```
std::map<T1, T2, std::greater<>> map_name;
```

Associative: std::set

- Unique values only.
- Default sorted in ascending order
- Usually implemented as a red-black tree.
- Logarithmic access time for most operations
 - #include <set> to use
 - Good as a fallback set type, but faster alternatives exist
 - Such as std::unordered_set

```
#include <set>

std::set<std::string> s;

s.insert("bat"); // The insert function takes in something convertible to the value.
s.emplace("cat"); // This is the preferred way of using a set (may reduce copying)

assert(s.contains("bat")); // true!

auto it = s.find("bat"); // Iterator to bat if present, otherwise s.end()

for (const std::string& v : s) {
    std::cout << s << '\n';
}
```

Descending order:

`std::set <T, std::greater<>> set_name;`

Unordered Associative Containers

Provide fast retrieval of data based on hashed keys.

<code>std::unordered_set</code>	A collection of unique keys.
<code>std::unordered_map</code>	A collection of key/value pairs
<code>std::unordered_multiset</code>	A collection of keys
<code>std::unordered_multimap</code>	A collection of unique keys to many values

All unordered associative containers are implemented as chained
hashTables.

Their interfaces are mostly the same as their ordered counterparts.



Associative: std::unordered_set/map

```
// declaring set for storing string data-type
std::unordered_set<std::string> string_set = {"C++", "Java"};

std::string key;
std::cout << "entered your preferred language";
std::cin >> key;

// find returns end iterator if key is not found,
// else it returns iterator to that key
if (string_set.find(key) == string_set.end()) {
    std::cout << key << " not found" << std::endl;
} else {
    std::cout << "Found " << key << std::endl;

// now iterating over whole set and printing its
// content
std::cout << "\nAll elements : ";

for (const auto elem : string_set) {
    std::cout << elem << std::endl;
}
}
```

- Implemented using a hash table where keys are hashed into buckets of a hash table so that the insertion is always randomised.
- Memoised constant time lookup operation
 - Operations takes constant time $O(1)$ on an average, linear time $O(n)$ in worst case.
 - Search, insertion, deletion are constant time.

Container Adaptors

For Sequential Containers, adaptors provide a restricted interface.

<code>std::stack</code>	Adapts a container to provide a stack (LIFO data structure)
<code>std::queue</code>	Adapts a container to provide a queue (FIFO data structure)
<code>std::priority_queue</code>	adapts a container to provide a priority queue (order depends on element priority)



Adaptors: std::stack

- vector or deque (by default) or list
 - Push(insert) pop(remove) only from back
- LIFO

```
#include <iostream>
#include <stack>
int main() {
    std::stack<std::string> st;
    st.push("C++");// same data which is written during declaration of stack
    st.push("Java");
    st.push("Python");
    st.push("MATLAB");
    st.pop();
    st.pop();
    st.pop();

    while (!st.empty()) {
        std::cout << st.top() << " ";
        st.pop();
    }
}
```



Container Performance

- Performance still matters.
- Standard containers are abstractions of common data structures.
- Different containers have different time complexities of the same operation (see right)
- cppreference has a summary of them [here](#).

Operation	vector	list	queue
container()	O(1)	O(1)	O(1)
container(size)	O(1)	O(N)	O(1)
operator[]()	O(1)	-	O(1)
operator=(container)	O(N)	O(N)	O(N)
at(int)	O(1)	-	O(1)
size()	O(1)	O(1)	O(1)
resize()	O(N)	-	O(N)
capacity()	O(1)		
erase(iterator)	O(N)	O(1)	O(N)
front()	O(1)	O(1)	O(1)
insert(iterator, value)	O(N)	O(1)	O(N)
pop_back()	O(1)	O(1)	O(1)
pop_front()		O(1)	O(1)
push_back(value)	O(1)+	O(1)	O(1)+
push_front(value)		O(1)	O(1)+
begin()	O(1)	O(1)	O(1)
end()	O(1)	O(1)	O(1)

Container-like Types

These types hold an element but differ in that they are specialised.

<code>std::pair</code> (2-tuple) <code>std::tuple</code> (n-tuple)	Heterogenous list of values
<code>std::function</code>	Holds 0 or 1 callables
<code>std::optional</code>	Contains 0 or 1 values of a type
<code>std::variant</code>	A type-safe tagged union
<code>std::any</code>	A type that can hold a value of any type.



Container-like: std::tuple

- Heterogenous list of types
- Access elements with std::get by position or by type
- #include <tuple> to use
- Useful when needing to pass around lots of disjoint data

```
#include <tuple> // #include<functional>
```

```
std::tuple<float, char> t1 = {3.14f, 'c'};  
auto t2 = std::make_tuple(3.14f, 'c'); // equivalent way to make a tuple
```

```
std::cout << std::get<0>(t1) << "\n"; // prints 3.14  
std::cout << std::get<char>(t2) << "\n"; // prints 'c';
```

Container-like: std::function

- general-purpose polymorphic function wrapper
- Encapsulates callables
- Usable like a regular function
 - #include <functional> to use
 - Useful when coding in a functional paradigm, such as partial application.

```
#include <functional>
using namespace std::placeholders; // for _1 in the std::bind example

int add(int n1, int n2) { return n1 + n2; }
std::function<int(int, int)> adder = add; // now adder is the same add()

std::function<int(int)> plus_one = std::bind(adder, 1, _1);
std::cout << plus_one(6770) << std::endl; // prints 6771
```

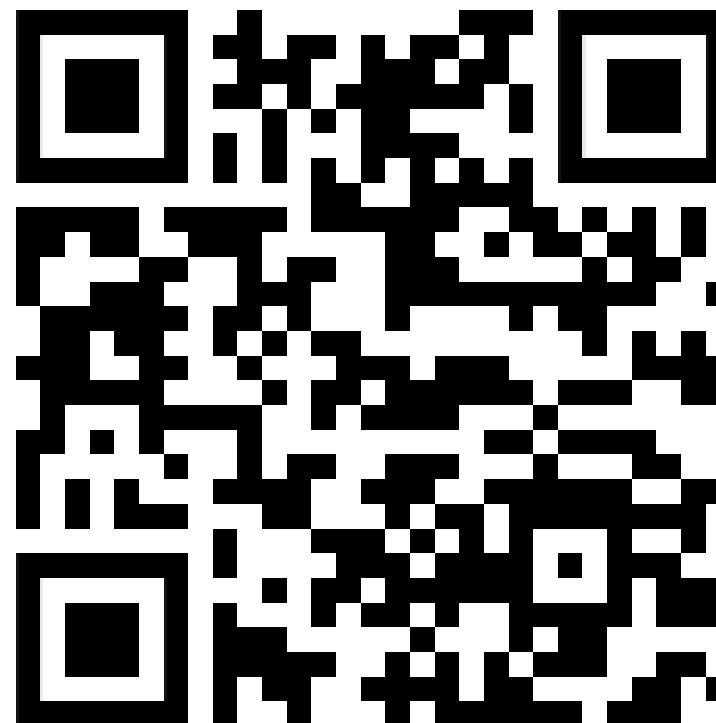
Container-like: std::optional

- A nullable type holding a value or nothing
- Type-safe: cannot access the element if it does not exist
- `#include <optional>` to use
- Useful when needing to convey the absence of a value.

```
#include <optional>

std::optional<double> divide (double top, double bot) {
    // this function has “no result” if the denominator is 0!
    return bot == 0 ? std::optional<double>{} : std::optional<double>{top / bot};
}
auto quotient = divide(5, 0);
std::cout << std::boolalpha << quotient.has_value() << std::endl; // prints false
```

Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

2.3 Standard Iterators



UNSW
SYDNEY

What is an Iterator?

- Iterators abstract the concept of a **Pointer**
 - i.e., reference semantics with the same or a subset of the operations as a pointer.
- Iterators are types that abstract container data as a linear **sequence** of objects.
- Iterators allow us to connect a wide range of containers with a wide range of algorithms via a common interface.
- Different categories of iterators - to support read, write and random access.
- Containers define their own iterators.



- `a.begin()`: abstractly "points" to the first element
- `a.end()`: abstractly "points" to one past the last element
- `a.end()` is not invalid itself, but it is illegal to dereference it.
- If `iter` is an iterator to the k -th element, then:
 - `*iter` is the k -th element.
 - `++iter` abstractly points to the $(k + 1)$ -st element

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> nums = {1, 2, 3};
    // could use std::vector<std::string>::iterator instead of auto
    for (auto iter = nums.begin(); iter != nums.end(); ++iter) {
        std::cout << *iter << "\n";
    }
}
```

- `a` is a container with all its n objects ordered

`a.begin()`

1st

2nd

...

nth

`a.end()`

Constant Iterator vs. const_iterator

- Remember pointers can be top-level or bottom-level const.
- A **constant iterator** is a *top-level const* iterator. The iterator variable cannot be reassigned.
- A `const_iterator` is a bottom-level `const` iterator. The element pointed to by the iterator cannot be modified.
- Just like a pointer, an iterator can both be a `constant iterator` and a `const_iterator` simultaneously.
 - Not very useful, however.

```
#include <vector>

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

// std::vector<int>::iterator
auto it = n.begin();

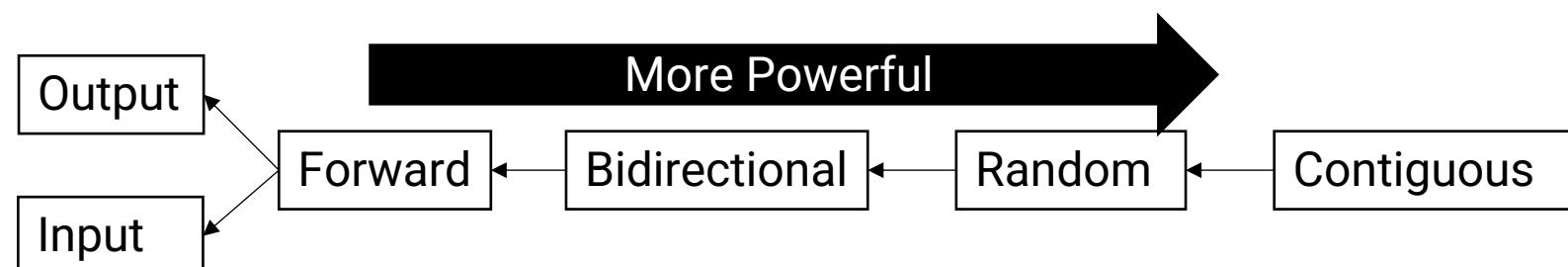
// const std::vector<int>::iterator;
const auto constant_it = n.begin();

// std::vector<int>::const_iterator
auto const_it = n.cbegin();

// const std::vector<int>::const_iterator
const auto constant_const_it = n.cbegin();
```

Iterator Categories

Operation	Output	Input	Forward	Bidirectional	Random Access	Contiguous
Read		<code>=*p</code>	<code>=*p</code>	<code>=*p</code>	<code>=*p</code>	<code>=*p</code>
Access		<code>-></code>	<code>-></code>	<code>-></code>	<code>->, []</code>	<code>->, []</code>
Write	<code>*p=</code>		<code>*p=</code>	<code>*p=</code>	<code>*p=, p[n]=</code>	<code>*p=, p[n]=</code>
Advance	<code>++</code>	<code>++</code>	<code>++</code>	<code>++, --</code>	<code>++, --, +, -, +=, -=</code>	<code>++, --, +, -, +=, -=</code>
Compare		<code>==, !=</code>	<code>==, !=</code>	<code>==, !=</code>	<code>==, !=, <, >, <=, >=</code>	<code>==, !=, <, >, <=, >=</code>
Array-like?	Potentially	Potentially	Potentially	Potentially	Potentially	Guaranteed



Iterator Invalidation

- An automatic process where an operation on a container makes use of a pre-existing iterator undefined behaviour.
- Read operations never invalidate iterators.
- Write operations *may* invalidate iterators:
 - Array-based containers (vector, deque) usually invalidate iterators after insertion and removal.
 - Associative containers' iterators are invalidated when an element is erased.
- More on this in Week 4.

```
auto vec = std::vector<int>{1,2,3,4,5};  
  
// push_back _invalidate container  
for (auto it = vec.begin();  
     it != vec.end();  
     it++) {  
    if (*it == 5) {  
        vec.push_back(2);  
    }  
} // this loop is ill-formed
```



Stream Iterators

- `iostreams` (`std::ifstream`, `std::ofstream`, `std::cout`, `std::cin`, etc.) do not define their own iterators.
- Can use `std::istream_iterator` or `std::ostream_iterator` to make one.
- Need to specify in the `<>` what the type of data to find is.
 - Delimits by default on whitespace.
 - Delimiter is customisable

```
auto in = std::ifstream("ass1.sol");
auto begin = std::istream_iterator<int>(in);

// below is equivalent to EOF for stream iters
auto end = std::istream_iterator<int>{};

std::cout << *begin++ << "\n"; // read an int
++begin; // skip the next int
for(; begin != end; ++begin) // read the rest
    std::cout << *begin << std::endl;
```

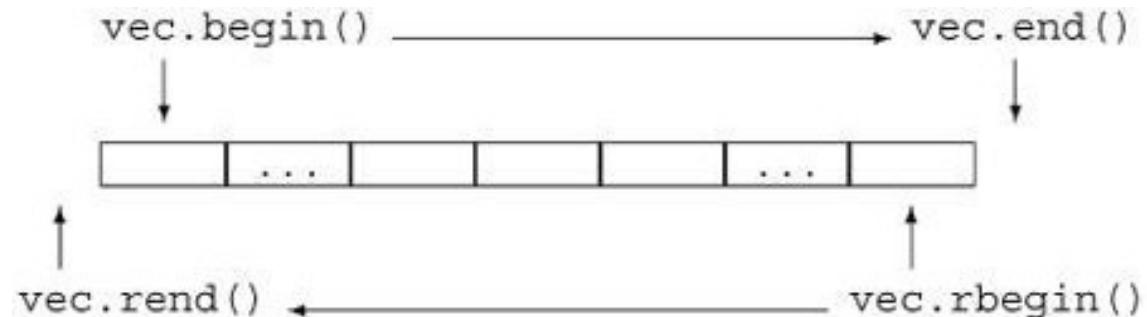
Iterator Adaptors

- The `<iterator>` header offers many convenience functions to adapt iterators for use in common problems.
- We will look at the most common ones:
 - Reverse iterators
 - Back-inserter iterators
 - Insert iterators



Reverse Iterators

- Reverse iterators allow us to iterate backwards through a container.
- All standard containers provide methods to get a reverse iterator.
- Later, we will see how to make our own reverse iterators extremely easily.



```
std::vector<int> v = {3, 6, 9};  
  
for (auto r = v.rbegin(); r != v.rend(); ++r) {  
    std::cout << *r << std::endl;  
} // prints 9, 6, 3
```



Back Inserter Iterators

- Gives you an output iterator for a container that supports `push_back()`.
- Writing to the output iterator causes the underlying container to push back the new element.

```
#include <iterator>

std::vector<int> nums;
auto it = std::back_inserter(nums);
*it = 42;
*it = 6771;

for (int i : nums) {
    std::cout << i << std::endl;
} // prints 42, 6771
```



Insert Iterators

- Gives you an output iterator for a container that supports `insert()`.
- Writing to the output iterator causes the underlying container to insert the new element.
- Need to give the container as well as an iterator into the container to start inserting from.

```
#include <iterator>

std::vector<int> v;
auto it = std::inserter(v, v.begin());
*it = 42;           // calls v.insert()
*it = 6771;         // calls v.insert()

for (int i : v) {
    std::cout << i << std::endl;
} // prints 42, 6771
```



Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

2.4 Standard Algorithms



UNSW
SYDNEY

Algorithms

- The Standard Library provides a plethora of algorithms that operate on iterators.
- In this way, they can work on a large number of containers as long as those containers can be represented via an appropriate iterator.
- These algorithms can be found amongst a few header files:
 - Majority are in `<algorithm>`
 - Some are in `<numeric>` (notably, `std::accumulate`)
 - The full list of algorithms can be found [here](#)



Simple Example

- What's the best way to sum a list of numbers?
 - Is it with a C-style for-loop?
 - Is it with an iterator-based for-loop?
 - Is it with a ranged for-loop?
- The best way is to use a Standard Algorithm:
`std::accumulate!`

```
#include <numeric>
auto v = std::vector<int>{42, 6771, 96};
int total =
    std::accumulate(v.begin(), v.end(), 0);
// contrast to...
int n = 0;
for (auto i = 0u; i < v.size(); ++i) {
    n += v[i];
}
```

std::accumulate Implementation

```
template <typename InputIt, typename T>
T accumulate(InputIt first, InputIt last, T n) {
    for (; first != last; ++first) {
        n += *first;
    }
    return first;
} // the underlying method of accumulate should be familiar
```

Almost all of the algorithms are implemented as **function templates** for maximum code reuse and efficiency.

At this point you do not need to know how to write a template.



Common Algorithms

- Some of the most commonly used algorithms are:
 - `std::copy` – a type-safe and more powerful replacement of `memcpy()`.
 - `std::find` – a linear search algorithm to find an element in a container.
 - `std::transform` – C++’s version of `map()` from other languages.
 - `std::swap` – a classic three-step swap implementation.
 - `std::accumulate` – performs a left fold. Can be used for sums, products, and other arbitrary operations.



Example: std::copy

```
#include <iostream>
#include <iterator>
#include <string>
int main() {
    std::copy(
        std::istream_iterator<std::string>(std::cin),
        std::istream_iterator<std::string>{},
        std::ostream_iterator<std::string>(std::cout)
    ); // this echoes each line of stdin to stdout a.k.a the cat command
}
```



Example: std::find

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};

    auto it = std::find(nums.begin(), nums.end(), 4);

    if (it != nums.end()) {
        std::cout << "Found it!" << "\n";
    }
}
```



Using Algorithms

- Some algorithms accept a predicate function that performs a task appropriate to the algorithm.
 - E.g., `std::find_if` accepts a function that says whether or not an element is “found”.
 - `std::accumulate` accepts a function that defines what operation to use instead of the default summation.
- Having to define a new function for a one-off operation makes using algorithms burdensome.
- C++11 introduced Lambda Functions to solve this problem.



Lambda Functions

- A function that can be defined inside of other functions.
- Can be passed to and returned from functions and stored in variables.
- Can capture all, none, or some of the variables in the enclosing scope.
- Convenient and replaces one-off functions

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

int main() {
    std::string s = "hello world";
    std::for_each( // modifies each element
        s.begin(),
        s.end(),
        [] (char& c) { c = std::toupper(c); }
    );
}
```



Anatomy of a Lambda

```
[capture] (parameters) -> optional_return_type {  
    body;  
}
```

```
[]{ /* lambdas with no parameters can omit () */};
```

```
// lambdas can be stored into variables. Type must be auto  
auto rand = []() -> int { return 6771; }
```

```
// lambdas with no captures decay into regular function pointers  
int(*cmp)(int, int) = [](int a, int b) { return a < b; };
```



Lambda Captures

- By default, lambdas execute in their own scope.
- Gain access to outside scope by capturing
 - Capture by value
 - Capture by reference
- Considerations:
 - Capturing by value makes a copy. This may be expensive for large types.
 - Capturing by reference could lead to dangling references if returning a lambda from a function

```
int a = 0, b = 2;  
  
// will not compile: did not capture a  
[]() { std::cout << a << std::endl; }  
  
// OK: captured a by value  
[a]() { std::cout << a << std::endl; }  
  
// OK: captured a by reference  
[&a]() { std::cout << a << std::endl; }  
  
// OK: captured everything by reference  
[&]() { std::cout << a + b << "\n"; }  
  
// OK: captured everything by value  
[=]() { std::cout << f + b << "\n"; }
```

Algorithms Performance & Portability

- Consider:
 - Number of comparisons for binary search on a vector is $O(\log N)$
 - Number of comparisons for binary search on a linked list is $O(N \log N)$
 - The two implementations are completely different
- We can call the same function on both of them
 - It will end up calling a function with two different overloads: one for a forward iterator, and one for a random access iterator
- Trivial to read
- Trivial to change the type of a container

```
#include <algorithm>
#include <list>
#include <vector>

int main() {
    // Lower bound does a binary search
    // and returns the first value >= the argument.
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::lower_bound(v.begin(), v.end(), 5);

    std::list<int> l {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::lower_bound(l.begin(), l.end(), 5);
}
```

Algorithms' Iterator Requirements

An **algorithm** requires certain kinds of iterators for its operation

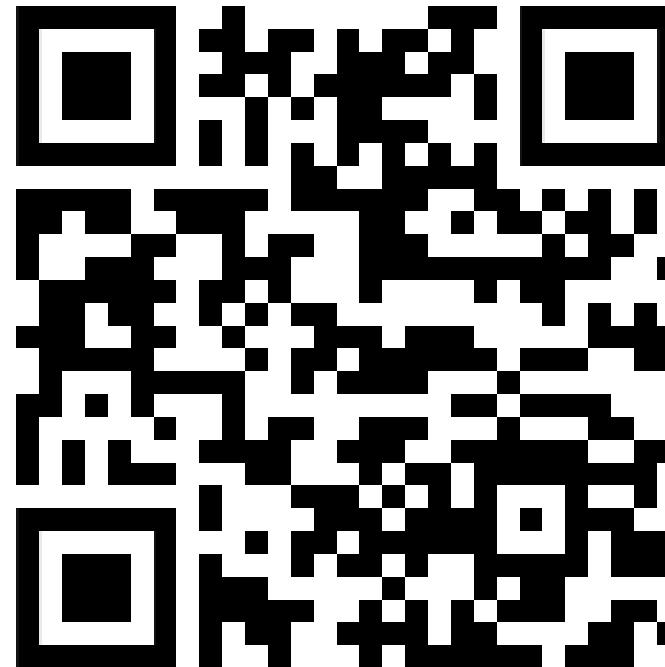
- **input**: find(), equal()
- **output**: copy()
- **forward**: binary_search()
- **bidirectional**: reverse()
- **random**: sort()

A **container's** iterator falls into a certain category

- forward**: forward_list
- bidirectional**: map, list
- random**: vector, deque

The container adaptors (stack, queue, etc.) do not have iterators

Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

3.1 - Scopes



UNSW
SYDNEY

What is a Scope?

- The scope of a variable is the part of the program where it is accessible
 - Scope starts at variable definition
 - Scope (usually) ends at the next }
 - You're probably familiar with this even if you've never seen the term
- Define variables as close to first usage as possible
- This may be the opposite of what you've been previously taught.
 - Defining all variables at the top is especially bad in C++

```
#include <iostream>

int i = 1;
int main() {
    std::cout << i << "\n"; // prints 1
    if (i > 0) {
        int i = 2;
        // int &ri = i; // create a reference
        std::cout << i << "\n"; // prints 2
    }
    int i = 3;
    std::cout << i << "\n"; // prints 3
}
std::cout << i << "\n"; // prints 2
}
std::cout << i << "\n"; // prints 1
```

New Scopes

- if-statements
- switch-statements
- Loops
- Compound statements
 - i.e., Random Braces
- Function bodies
- Class types
- Namespaces
- Global scope

```
int i; // global scope

namespace N { // namespace scope
    int j = i;
}

struct point { // struct scope
    int x;
    int y;
};

int main() { // new function scope
    if (int k = i; k >= 0) {
        // new if-statement scope
    }
    for (int m = 0; m < N::j; ++m) {
        // new loop scope
    }
}
```



Namespaces

- Used to:
 - Group related names together
 - Modularise code
- Can only be created:
 - At global scope
 - At namespace scope (nested namespace)
 - Entities inside accessed with the scope operator ::

```
// in std.h
namespace std {
    auto cout = /* definition */;
    char get_char();
}

// in std.cpp
namespace std {
    char get_char() { return 'c'; }
}

// main.cpp
int main() {
    std::cout << std::get_char() << "\n";
}
```



Nested Namespaces

- It is possible to define a namespace within another namespace.
 - The Standard Library does this with `std::chrono` and `std::ranges`
- Prefer top-level and occasionally two-tier namespaces to multi-tier.
- It's okay to own multiple namespaces per project if they logically separate things.
- Nested entities accessed by chaining ::

```
// in std.h
namespace std {
    auto cout = /* definition */;
    namespace chars {
        char get_char();
    }
}

// in std.cpp
// legal syntax since C++17
namespace std::chars {
    char get_char() { return 'c'; }
}

// main.cpp
int main() {
    std::cout << std::chars::get_char() << "\n";
}
```



Inline Namespaces

- It is possible to define nested namespaces as being `inline`.
- This injects all names inside the nested namespace into the enclosing one.
- Can still access names via the fully-qualified name.
- Useful for symbol versioning:
 - "Default" version of a type, etc., accessible through the enclosing namespace.
 - Other versions possible through nested namespaces.
 - Change what is the "current" symbol by moving it into the inline namespace.

```
namespace std {
    inline namespace curr {
        class vector {
            /* current interface of vector */
        }
    }
    namespace cpp98 {
        class vector {
            /* old interface of vector */
        }
    }
}

std::vector v1; // default: std::curr::vector
std::curr::vector v2; // same as above
std::cpp98::vector v3; // opt-in to old version
```



Anonymous Namespaces

- In C you had *static* functions that made functions local to a file.
- C++ uses anonymous namespaces to achieve the same effect.
- Functions that you don't want in your public interface should be put into anonymous namespaces.
 - Defining an anonymous namespace in a header file will still make those public though!
 - Any names inside an anonymous namespace are injected into the enclosing scope automatically.

```
// in api.h
namespace api {
    char get_char();
}

// in api.cpp
namespace {
    // helper only accessible in this file
    char helper() { return 'c'; }
}

namespace api {
    char get_char() { return helper(); }
}

// main.cpp
int main() {
    std::cout << api::get_char() << "\n";
}
```



Namespace Aliases

- Gives a namespace a new name.
- Often good for shortening nested namespaces.
- Aliases should be descriptive.

```
namespace chrono = std::chrono;  
namespace views = ranges::views;
```



Namespace using-Directives

- It is possible to inject all or some names in a namespace into the enclosing scope using a `using` directive.
- This should be avoided as much as possible to prevent name collisions.
- **DO NOT** write in header files `using namespace std;`
- Limiting the scope of `using-directives` to inside functions is OK.

```
// potentially catastrophic
// using namespace std;

int main() {
    using std::vector;
    vector<int> v = {1, 2, 3};

    if (v.size()) {
        using namespace std::chrono;
        auto txt = millisecond{500};
    } else {
        // this won't work: different scope
        auto txt2 = millisecond{500};
    }
}
```



Name Lookup

- There are certain complex rules about how overload resolution works that will surprise you.
- This is called **Argument-Dependent Lookup (ADL)** and will not be covered in this course.
- It is best to always fully-qualify your function calls.
- Even if you're within the same namespace!

```
namespace ex {  
    struct empty {};  
    void f(empty) { std::cout << "hi\n"; }  
}  
  
void f(ex::empty) { std::cout << "hi\n"; }  
  
int main() {  
    ex::empty e;  
  
    f(e); // which function is called?  
    // due to ADL, both ::f and ex::f are  
    // equally callable! This is ambiguous!  
  
    ex::f(e); // OK! Calls ex::f  
    ::f(e); // OK! Calls the global f  
}
```



Object Lifecycle in C++

- An object is a piece of memory of a specific type that holds some data.
 - All variables are objects.
 - This does not add overhead because objects are stack-allocated by default.
- Object lifetime starts when it comes in scope.
 - Memory for the object is first allocated.
 - Then it is formally “constructed”.
 - Every type has at least one constructor that says how to initialise it.
- Object lifetime ends when it goes out of scope.
 - The object is formally “destructed”.
 - The memory is then deallocated.
 - Every type has exactly one destructor that knows how to destruct an object.



Object Construction

- We generally use () to call functions, and {} to construct objects.
 - Functions only callable with ()
 - Construction can use () or {}
- There are cases when which is used matters:
 - Sometimes it is ambiguous between a constructor and an initialiser list.
 - See the [most vexing parse](#)
- Fundamental types have constructors too.
 - They do *nothing* by default.
 - Initialisation must be manually performed by the programmer.
 - Especially problematic with pointers.

```
void ex() {  
    int i; // default ctor: uninitialized  
    int i{}; // initialised with 0.  
    int i{4}; // value-initialised to 4  
    int i(4); // same as above.  
  
    // is {1, 2} an initialiser list?  
    // or a call to one of vector's constructors?  
    // this will be interpreted as an init list  
    std::vector<int> v{1, 2};  
  
    int *p; // default ctor: uninitialized.  
    // p is known as a "wild" pointer  
}
```



Object Destruction

- Objects are destructed once they go out of scope.
- An object goes out of scope when:
 - Local variables: the } marking the end of its scope is encountered.
 - static/global variables: when the program ends
 - Dynamically-allocated variables: when the programmer calls `free()`/`delete`
- Destruction is deterministic
 - C++ is not garbage-collected.

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

int main() {
    int *i = new int{4};

    {
        std::string str = "hi!";
    } // str is destructed

    delete i; // the int is destructed.
}

// i would NOT have been destructed
// at the end of main since it is
// dynamically allocated.
// It is our responsibility to delete it.

// v is destructed here at program end
```



Why is the Object Lifecycle Useful?

Can you think of a thing where you always have to remember to do something when you're done?

- What happens if we forget to close a file?
- How easy to spot is the mistake?
- How easy would it be for a compiler to spot this mistake for us?
 - How would it know?
 - Through the object lifecycle rules!



Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

3.2 – Class Types



UNSW
SYDNEY

What is Object-Oriented Programming?

- A class uses data abstraction and encapsulation to define an abstract data type:
 - Abstraction: separation of interface from implementation
 - Useful as class implementation can change over time
 - Encapsulation: enforcement of this via information hiding
- This abstraction leads to two key concepts:
 - Interface: the operations used by the user (an API)
 - Implementation: the data members the bodies of the functions in the interface and any other functions not intended for general use



C++ Class Types

A **class type** in C++ is defined as one of the below three entities:

- **structs**
 - Same as they are in C.
 - A structure made up of data of any type.
- **unions**
 - Same as they are in C.
 - We will not be covering unions much in this course.
- **classes**
 - New in C++.
 - Classes are virtually identical to structs.



C++ Classes

Since you've completed COMP2511 (or equivalent), C++ classes should be straightforward and at a high-level follow very similar principles.

- A class:
 - Defines a new type
 - Is created using the keyword `class`
 - May define some members (methods, data)
 - Contains zero or more public, protected, or private sections
 - Is instantiated through a constructor.
- A member function:
 - must be declared inside the class
 - may be defined inside the class (it is then inline by default)
 - may be declared `const`, when it doesn't modify the data members
- The data members should be private, representing the state of an object.



Classes vs. Structs

- A class and a struct in C++ are almost exactly the same.
- The only technical difference is that:
 - All members of a struct are public by default
 - All members of a class are private by default
- Structs generally are used for simple types with few or no methods.
 - These kinds of structs are called PODs: plain old data.
- Intended use of your type should determine whether it is a struct or class

```
// default everything is private
class foo {
    int member_;
};

// default everything is public
// we often times denote this with
// a different style of member naming
// note the lack of a trailing -
struct footwo {
    int member;
}
```

Member Access Control

- This is how we support encapsulation and information hiding in C++.
- Can have multiple sections of the same **access specifier**.
- They are:
 - public – accessible by everyone
 - protected – accessible by children of the class and the class itself.
 - private - accessible only by this class
 - virtual - cover this in the polymorphism module.

```
class foo {  
    // Members accessible by everyone  
public:  
    foo(); // The default constructor.  
  
    // Accessible by this class & children.  
    // Will discuss more later in the course  
protected:  
  
    // Accessible only by this class  
private:  
    void private_member_function();  
    int private_data_member_;  
  
    // Can define the same access specifier again  
public:  
};
```



Constructor

- Constructors define how this object should be initialised.
- A constructor has the same name as the class and no return type.
- The constructor that is callable with zero arguments is called the **default constructor**.

```
#include <iostream>

class my_class {
public:
    // this is a constructor
    my_class(int i) {
        i_ = i;
    }
    get_val() {
        return i_;
    }

private:
    int i_;
};

int main() {
    auto mc = my_class{1};
    std::cout << mc.get_val() << "\n";
}
```



Construction Order

Class construction follows this pseudocode algorithm.

```
for each data member in declaration order
    if it has a user-defined initialiser
        Initialise it using the user-defined initialiser
    else if it is a fundamental type (integral, pointer, bool, etc.)
        do nothing (leave it as whatever was in memory before)
    else
        Initialise it using its default constructor
```



Member Initialiser List

- The initialisation phase occurs before the body of the constructor is executed, regardless of whether a member initialiser list is supplied.
- You are able to provide initial values for data members before entering the constructor body.
- This avoids initialising a data member only to have its value overwritten.
- You should **always** use a member initialiser list.
- You also must ensure you initialise data members in declaration order.
 - Otherwise the behaviour is undefined.

```
class user {  
public:  
    user(std::string name, int age)  
        // everything following the colon  
        // is the member initialiser list  
    : name_{name}, age_{age} {  
        /* more complex set-up */  
    }  
  
private:  
    std::string name_;  
    int age_;  
};
```



Delegating Constructor

- A constructor may call another constructor from the member initialiser list.
- Since the other constructor must construct all the data members, you should not specify anything else in the list.
- The other constructor is called completely before this one.
- This means the lifetime of the object has begun!

```
class point2d {  
public:  
    // the default ctor delegates to the  
    // 2-arg constructor, supplying  
    // default values  
    point2d() : point2d(0, 0) {}  
  
    point2d(float x, float y)  
        : x_{x}, y_{y} {}  
  
private:  
    float x_;  
    float y_;  
};
```



In-class Initialisers

- It can be easy to forget to initialise all data members, especially when there are many.
- You can use in-class initialisers as a last resort fallback if a data member is not initialised with a member initialiser list.
- It is best to minimise their use, as having initialisation logic spread out can be hard to read.

```
class point2d {  
public:  
    point2d(float x, float y)  
        : x_{x}, y_{y} {}  
  
private:  
    // with no default constructor,  
    // the in-class initialisers  
    // will be used instead.  
  
    float x_ = 0;  
    float y_ = 0;  
};
```

explicit

- If a constructor for a class is callable with a single parameter, the compiler will create an implicit type conversion from the parameter to the class type.
- This **may** be the behaviour you want
 - But probably not
 - You have to opt-out of this implicit type conversion with the **explicit** keyword.

```
class point2d {  
public:  
    // single arg parameter  
    explicit point2d(float n)  
        : x_{n}, y_{n} {}  
  
    // this ctor is also callable with a single arg  
    explicit point2d(float *f, int n = 2)  
        : x_{f[0]}, y_{n >= 2 ? f[1] : f[0]} {}  
  
private:  
    float x_;  
    float y_;  
};  
  
point2d ex() {  
    // error: single arg constructor is explicit  
    return 3.0f  
  
    float arr[2] = {0.0f};  
    // error: 2-arg constructor also is explicit  
    return arr;  
}
```



Destructor

- Called when the object goes out of scope.
- How would you use one?
 - Freeing malloc'ed memory.
 - Closing files.
 - Unlocking mutexes.
 - Aborting database transactions.
 - Any kind of resource clean-up.
- noexcept is part of C++ exceptions (we will cover this later)

```
// from Java: a boxed integer
class integer {
public:
    integer(int i) : ptr_{new int{i}} {}

    // never forget to free a pointer
    // ever again!
    ~integer() {
        delete ptr_;
    }

private:
    int *ptr_;
}
```



Special Member Functions

- There are six special member functions:
 - The default constructor
 - The copy constructor
 - The move constructor
 - The destructor
 - The copy-assignment operator
 - The move-assignment operator

```
class foo {  
public:  
    // default constructor  
    foo();  
  
    // copy/move (cover later) constructor  
    foo(const foo &other);  
    foo(foo &&other);  
  
    // destructor  
    ~foo();  
  
    // copy/move (cover later) operators  
    foo &operator=(const foo &other);  
    foo &operator(foo &&other);  
};
```



Synthesised Special Members

- The compiler is able to *synthesise* definitions for the special member functions if the user does not provide one.
- The synthesised version:
 - Default construction: tries to default construct all the data members in turn
 - Copy construction: calls the copy constructor of each data member
 - Destruction: goes in *reverse-order*, destructing each data member.
 - Etc.
- It is possible to opt-*into* synthesis with `default`.
- Likewise, it is possible to opt-out of synthesis with `delete`

```
class point2d {  
public:  
    // even one user-supplied ctor stops  
    // the compiler-generated default.  
    point2d(float a, float b);  
    point2d() = default; // opt back into it  
  
    // now this class is no longer copyable  
    point2d(const point2d &other) = delete;  
  
    // ensure compiler-synthesised dtor.  
    ~point2d() = default;  
  
private:  
    float x_;  
    float y_;  
};
```



Removing Unneeded Special Members

- There are several special functions that we must consider when designing classes.
- Ask yourself the question: does it make sense to have this default member function?
 - Yes: Does the compile synthesised function make sense?
 - No: write your own definition
 - Yes: write "<function declaration> = default;"
 - No: write "<function declaration> = delete;"



Non-Static Members

- A class can have member functions (methods).
- A class can also have data members.
- The size of a class only depends on the types and declaration order of its members.
 - Due to alignment requirements, the compiler may insert padding into your class.
 - `sizeof(class) >= 1` (why?)

```
class foo {  
public:  
    void speak();  
private:  
    int i;  
    void *ptr;  
    bool b;  
};  
  
class foo2 {  
public:  
    void speak();  
private:  
    bool b;  
    int i;  
    void *ptr;  
};  
  
static_assert(sizeof(foo) == 24); // why?  
static_assert(sizeof(foo2) == 16); // why?
```



Incomplete Types

- An incomplete type is a type that has been declared but not defined.
- You can only form pointers and references to incomplete types.
 - Except void& is illegal.
- Because of this restriction, a class cannot have data members of its own type.
- Since a class is considered declared once its name has been seen, it can have pointer/reference members to its own type.

```
struct node {  
    int data;  
  
    // node is incomplete.  
    // This is invalid.  
    // This would also make no sense.  
    // What is sizeof(Node)??  
    node next;  
  
    // this, however, is fine.  
    node *next;  
  
    // this is fine, too.  
    node &next;  
};
```



The this Pointer

- Member functions have an extra implicit parameter, named `this`.
- This is a pointer to the object on behalf of which the function is called.
- A member function does not explicitly define it, but may explicitly use it
 - As of C++23, you may now define it too.
- The compiler treats an unqualified reference to a class member as being made through the `this` pointer.
- Generally we use a `"_"` suffix for class variables rather than `this->` to identify them
- It is possible to overload a method based on the constness of `this`.

```
class point2d {  
public:  
    point2d(float x, float y) : x_{x}, y_{y} {}  
  
    float x() {  
        // the signature of this method is really  
        // float(point2d * const this)  
        return x_;  
    }  
  
    float x() const {  
        // the signature of this method is really  
        // float(const point2d * const this)  
        return x_;  
    }  
private:  
    float x_ = 0;  
    float y_ = 0;  
};
```



const Objects

- Member functions are by default only callable by non-const objects.
- You can declare a const member function which is callable by both const and non-const objects
- A const member function may only modify **mutable** members
 - There are **very few** use cases for mutable
 - One example is as a cache
 - A mutable member *should* mean that the physical state of the member can change without the logical state of the object changing.
 - In practice, this isn't always the case.

```
class point2i {  
public:  
    point2d(int a, int b) : x_{a}, y_{b} {}  
  
    const int& x() const { return x_; }  
    int &x() { return x_; }  
  
    const int& y() const { return y_; }  
    int &y() { return y_; }  
  
private:  
    int x_;  
    int y_;  
}  
  
const auto p = point2i{1, 2};  
  
p.x(); // OK! const-qualified method called  
p.y() = 4; // error: calls a non-const method
```

Static Members

- Static members belong to the class, as opposed to any particular object.
- Static methods are callable without any instance.
- Static methods are never const-qualified.
- Static data members' lifetime ends when the program ends.
- Use static members when something is associated with a class, but not a particular instance.

```
class point2d {  
public:  
    static point2d make_point(float a, float b) {  
        return point2d(a, b);  
    }  
  
    ~point2d() {  
        n_live_ -= 1;  
    }  
  
private:  
    point2d(float a, float b) : x_{a}, y_{b} {  
        n_live_ += 1;  
    }  
  
    inline static int n_live_ = 0; // since C++17  
  
    float x_;  
    float y_;  
};
```



Friends

- A class may declare another function or class as a friend.
 - Note: this does not declare the friend class or function itself.
 - It only says *if* such a class or function exists, then it is a friend.
- Friends are able to access the private members of the class.
- For a class C and friend F:
 - F is not a friend of the children of C.
 - F is not a friend of classes C is a friend of.
 - F is not a friend of any parent classes of C.
- Friends are *always* public.

```
class point2d {  
public:  
    /* Other implementation details... */  
    // declare distance as a friend of point2d  
    friend float dist(point2d &, point2d &);  
  
private:  
    float x_;  
    float y_;  
}  
  
float dist(point2d &l, point2d &r) {  
    // because dist is a friend, it can access  
    // the private members of point2d  
    return std::sqrt(l.x_ * r.x_ + l.y_ * r.y_);  
};
```



Hidden Friends

- It is possible to declare and define a friend inline in a class.
- This “hidden” friend has different look-up rules than usual:
 - Only discoverable through ADL
 - With a separate declaration outside the class, they are discoverable without ADL.
- Mostly used with operator overloading.

```
namespace hf {  
    class point2d {  
        public:  
            // other implementation details...  
  
            friend float dist(point2d &l, point2d &r) {  
                return std::sqrt(l.x_ * r.x_ + l.y_ * r.y_);  
            }  
  
        private:  
            float x_, y_;  
        };  
  
        float dist(point2d &, point2d &);  
    }  
  
    void ex() {  
        hf::point2d p = {1, 2}, q = {3, 4};  
  
        // OK: hidden friend found through ADL  
        dist(p, q);  
  
        // OK: found through 2nd declaration outside class  
        hf::dist(p, q);  
    }  
}
```

The Power of Friendship

Friendship seems to break encapsulation. Why use it?

- In general, friendship should be used sparingly.
- Friendship can be used between two interconnected classes.
 - E.g., a container and its iterator.
- Friendship can also be used for **API Extension**.
 - This does not mean unrelated code can extend an API.
 - It is for a different calling style for a logical operation of a type.
 - E.g., for a `vec3` class, the `cross()` operation should be a friend since `cross(a, b)` makes more sense than `a.cross(b)`.
- For certain operator overloads, it is more convenient for them to be defined as non-member friend functions.



Interface & Implementation

- We are used to declaring functions in header files and defining them in .cpp files.
- With classes we have two options:
 - Define the class completely in the header.
 - Define the class in the header but only declare methods. Define methods in the .cpp file.
- Defining everything in the header is easier for us but can slow down compilation due to duplicate code.
- Either way, static data members must be defined in a .cpp file unless declared inline.

```
// point.hpp - interface file
class point2d {
public:
    point2d(float a = 0, float b = 0);

    float &x() { return x_; }

private:
    static int cnt;
    float x_;
    float y_;
};

// point.cpp - implementation file
int point2d::cnt = 0; // static member

point2d::point2d(float a, float b)
: x_{a}, y_{b} {}           // ctor implementation
```



Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

4.1 – Operator Overloading



UNSW
SYDNEY

In this Lecture

Why?

- Operator overloads allow you to decrease your code complexity and utilise well defined semantics.

What?

- *Compile time polymorphism in existing operators.*
- Many different types of operator overloads.



Operator Overload Design

- Member operator overloads sometimes require **two** versions:
 - A non-const overload
 - A const overload
- A common example is `operator[]`:
 - The non-const overload is for setting values, e.g., `my_class[i] = 4;`
 - The const overload is for getting values
- Non-member operator overloads can *optionally* be friends.
 - If the operator overload can be implemented purely in terms of the class's **public** interface, it does not need to be a friend.
 - If it needs access to private or protected members, it should be a friend.
 - Non-member friend operator overloads should be **hidden friends**, i.e., defined inline in the class.
- In general, if an operator acts on a *particular* instance, it is a member function
 - Otherwise, it is a non-member function.



Motivating Example 1

- The last line is our best attempt to “Print the sum of two points”.
- `print(std::cout, point::add(p, q));`
- This is clumsy (and *ugly!*)
- We’d prefer to be able to write:
 - `std::cout << p + q;`

```
class point {  
public:  
    // implementation details...  
    static point add(const point &p, const point &q);  
    friend void print(std::ostream &, const point&);  
private:  
    int x_;  
    int y_;  
}  
  
point point::add(const point &p, const point &q) {  
    return point{p.x_ + q.x_, p.y_ + q.y_};  
}  
  
void print(std::ostream &os, const point &p) {  
    os << "(" << p.x_ << "," << p.y_ << ")";  
}  
  
point p = {1, 2}, q = {3, 4};  
  
print(std::cout, point::add(p, q)); // EW!
```

Motivating Example 2

- **Using operator overloading:**
 - Allows us to reuse our intuition with operators to implement nicer semantics for our classes!
 - Gives us a common and simple interface to implement classes to behave like built-in types.

```
class point {  
public:  
    // implementation details...  
    friend  
    point operator+(const point &p, const point &q);  
  
    friend  
    void operator<<(std::ostream &os, const point&p);  
private:  
    int x_;  
    int y_;  
}  
  
point operator+(const point &p, const point &q) {  
    return point{p.x_ + q.x_, p.y_ + q.y_};  
}  
  
void operator<<(std::ostream &os, const point &p) {  
    os << "(" << p.x_ << "," << p.y_ << ")";  
    return os;  
}  
  
point p = {1, 2}, q = {3, 4};  
  
std::cout << p + q << std::endl; // excellent
```



Operator Overloading in C++

- C++ supports a rich set of operator overloads.
- All operator overloads must have at least one operand of its type.
- Advantages:
 - Readability & reuse of existing code semantics.
 - Flexible and easy to maintain different operations.
 - No verbosity required for simple operations.
- Disadvantages:
 - Easy to abuse and create an overload that is distinct from its original meaning.
 - Slower than built-in operators due to a function call vs. a CPU instruction.
 - Only create an overload if your type has a single, obvious meaning for an operation.



Operator Overload Canonical Implementations

Type	Operator(s)	Canonical Implementation
I/O	<<, >>	Non-member function
Arithmetic	+, -, *, /, %, + (unary), - (unary)	Non-member function
Bitwise	&, , ^, ~, >>, <<	Non-member function
Compound Assignment	+ =, - =, * =, / =, % =, ^ =, =, & =	Member function
Comparison	>, <, >=, <=, !=, ==, <=>	Member or non-member
Logical	&&, , !	Non-member function
Assignment	=	Member function
Subscript	[]	Member function
Increment / Decrement	++ / -- (pre), ++ / -- (post)	Member function
Member Access & Dereference	->, *	Member function
Type Conversions	static_cast<type>	Member function
Call Operator	()	Member function



Overload: I/O

- Scope to overload for different types of output and input streams.
- Not member functions!
 - Why?
- `operator>>` takes a non-const reference to a point
 - Why?
- Returns a reference to an `iostream`.
 - Why?

```
struct point {  
    int x;  
    int y;  
  
    friend std::ostream &  
    operator<<(std::ostream &os, const point &p) {  
        os << "(" << p.x << "," << p.y << ")";  
        return os;  
    }  
  
    friend std::istream &  
    operator>>(std::istream& is, point& p) {  
        is >> p.x >> p.y;  
        return is;  
    }  
};  
  
int main() {  
    point p = {1, 2};  
    std::cout << p << std::endl;  
}
```

Overload: Arithmetic

- Classes that model mathematical objects often have the arithmetic operators defined.
- If you define one, it is best to define all of them.
- All operators return a new object.

```
class vec2 {  
public:  
    // other implementation details  
  
    friend  
    vec2 operator+(const vec2 &u, const vec2 &v) {  
        return vec2{u.x_ + v.x_, u.y_ + v.y_};  
    }  
  
    friend  
    vec2 operator-(const vec2 &u, const vec2 &v) {  
        return vec2{u.x_ - v.x_, u.y_ - v.y_};  
    }  
  
    // etc. for the other operators  
  
private:  
    double x_;  
    double y_;  
};
```



Overload: Bitwise

- The bitwise operators can be overloaded.
- Useful for making enum classes act like bit-flags.

```
enum class flag : int {  
    O_NONE  = 0b000,  
    O_READ   = 0b001,  
    O_WRITE  = 0b010,  
    O_RW     = 0b100  
};  
  
flag  
operator|(const flag &f1, const flag &f2) {  
    int f1i = static_cast<int>(f1);  
    int f2i = static_cast<int>(f2);  
  
    return static_cast<flag>(f1i | f2i);  
}  
  
flag  
operator&(const flag &f1, const flag &f2) {  
    int f1i = static_cast<int>(f1);  
    int f2i = static_cast<int>(f2);  
  
    return static_cast<flag>(f1i & f2i);  
}  
  
flag my_flags = flag::O_READ | flag::O_WRITE;
```



Overload: Compound Assignment

- Each class can have any number of **operator+=** operators, but there can only be one operator+=X (where X is a type).
- That's why in this case we have two multiplier compound assignment operators.

```
class vec2 {  
public:  
    // other implementation details  
  
    vec2 &operator+=(const vec2 &v) {  
        x_ += v.x_;  
        y_ += v.y_;  
        return *this;  
    }  
  
    // how would these be implemented?  
    vec2 &operator-=(const vec2 &v);  
    vec2 &operator*=(double scale);  
    vec2 &operator*=(int scale);  
    vec2 &operator/=(double scale);  
  
private:  
    double x_;  
    double y_;  
};
```



Overload: Comparisons

- Most types should implement at least the equality (`==`, `!=`) operators.
- If the type has an *ordering*, then it should implement the relational (`<`, `<=`, `>`, `>=`) operators too.
- This is largely a hassle.
 - Minimum functions to write are `operator==` and `operator<`.
 - The other four operators can be written in terms of these two.
 - C++ introduced the **spaceship** operator to solve this.

```
class vec2 {  
public:  
    // other implementation details  
  
    double x() const { return x_; }  
    double y() const { return y_; }  
  
private:  
    double x_;  
    double y_;  
};  
  
bool operator<(const vec2 &l, const vec2 &r);  
bool operator>(const vec2 &l, const vec2 &r);  
bool operator<=(const vec2 &l, const vec2 &r);  
bool operator<=(const vec2 &l, const vec2 &r);  
bool operator==(const vec2 &l, const vec2 &r);  
bool operator!=(const vec2 &l, const vec2 &r);
```



Overload: Spaceship Operator

- New in C++20: three-way comparison with operator`<=`.
 - If $a < b$, $(a \leqslant b) < 0$
 - If $a > b$, $(a \leqslant b) > 0$
 - If a is equivalent or equal to b , $(a \leqslant b) == 0$
- $a \leqslant b$ returns one of three kinds of *orderings*:
 - `std::strong_ordering`
 - `std::weak_ordering`
 - `std::partial_ordering`
- All orderings support “less than” and “greater than”.
- Only `std::strong_ordering` supports equality (if $a == b$ and $b == c$, $a == c$).
- `std::weak_ordering` and `std::partial_ordering` support “equivalence” (a is neither less than nor greater than b , but not equal).
- Only `std::partial_ordering` supports incomparable values ($a \leqslant b$ is always false).

Ordering	Equivalent values are...	Incomparable values are...
<code>std::strong_ordering</code>	Indistinguishable	Disallowed
<code>std::weak_ordering</code>	Distinguishable	Disallowed
<code>std::partial_ordering</code>	Distinguishable	Allowed

Comparison Category	Use Cases
<code>std::partial_ordering::less</code> <code>std::partial_ordering::equivalent</code> <code>std::partial_ordering::greater</code> <code>std::partial_ordering::unordered</code>	<ul style="list-style-type: none">• Floating-point numbers• Complex numbers• 2D points
<code>std::weak_ordering::less</code> <code>std::weak_ordering::equivalent</code> <code>std::weak_ordering::greater</code>	<ul style="list-style-type: none">• Case insensitive strings
<code>std::strong_ordering::less</code> <code>std::strong_ordering::equal</code> <code>std::strong_ordering::greater</code>	<ul style="list-style-type: none">• Integral types• Strings• 1D arrays



Spaceship Operator Example

```
#include <compare> // needed for std::partial_ordering

class point {
public:
    // other implementation details

    friend std::partial_ordering
    operator<=>(const point &p1, const point &p2) {
        auto x_ord = p1.x_ <=> p2.x_;
        auto y_ord = p1.y_ <=> p2.y_;

        return x_ord == y_ord ? x_ord : std::partial_ordering::unordered;
    }

private:
    int x_;
    int y_;
};
```



Default Comparisons (since C++20)

- `operator<=` is called whenever objects are compared with a relational operator.
- `operator==` is called when objects are compared with an equality operator.
- It is possible to default these member functions since C++20.
 - Default behaviour compares members in declaration order using the appropriate operator.

```
struct point2 {  
    double x;  
    double y;  
  
    auto // could also be a friend  
    operator<=(const point2 &) const = default;  
  
    // ALL other comparison operators  
    // automatically synthesised  
};  
  
struct vec2 {  
    double x;  
    double y;  
  
    bool // must be bool  
    operator==(const vec2 &) const = default;  
  
    // only operator==, operator!= synthesised  
};
```



Overload: Logical Operators

- Logical AND, OR, and NOT can be overloaded.
- They will **lose** their short-circuit behaviour if overloaded.
- Not a common overload.

```
class Bool { // a boxed bool
public:
    // other implementation details

    friend bool
    operator&&(Bool a, Bool b) {
        return Bool{a.b_ && b.b_};
    }

    // similar for the other operators
    friend bool operator||(Bool a, Bool b);
    friend bool operator!(Bool a, Bool b);

private:
    bool b_;
};
```

Overload: Assignment

- Assignment operator is one of the special member functions.
- Need to be careful to avoid self-assignment.
- Can be defaulted or deleted.
 - Default behaviour is to go through all members in declaration order and try to assign them.
- **Two kinds of assignment operators**
 - Copy assignment
 - Move assignment (will discuss later)

```
struct vec2 {  
    int x;  
    int y;  
  
    // canonical copy assignment signature  
    // defaulted: the compiler generates it  
    // will try to copy assign v.x to x  
    // and v.y to y.  
    vec2 &operator=(const vec2 &v) {  
        if (&v != this) {  
            // do this copy  
        }  
    }  
  
    // canonical move assignment operator  
    // don't need to worry about this yet.  
    vec2 &operator(vec2 &&v) = delete;  
};
```



Overload: Subscript

- Usually only defined on indexable containers.
- Need two overloads for get & set.
- During development, asserts can be used for bounds checking:
 - In other containers (e.g. vector), invalid index access is undefined behaviour.
 - Usually an explicit crash is better than undefined behaviour.
 - Asserts are stripped out of optimised builds, so no performance penalty.
- The getter version can either return by value or const reference.
 - If copying is expensive, const reference is preferred.

```
#include <cassert>

struct vec2 {
    int x;
    int y;

    int &operator[](int n) {
        assert(n == 0 || n == 1);
        return n == 0 ? x : y;
    }

    int operator[](int n) const {
        assert(n == 0 || n == 1);
        return n == 0 ? x : y;
    }
};
```

Overload: Increment & Decrement

- Prefix:
 - `++x`
 - `--x`
 - Returns a reference
- Postfix:
 - `x++`
 - `x-`
 - Returns a copy
- Need two overloads for pre- vs. post-fix.
 - Use an anonymous int variable in the **postfix** overload.
 - It is only used for overload resolution.
- Performance: prefix > postfix.

```
class tick {  
public:  
    // other implementation details  
  
    tick &operator++() {  
        cnt_++;  
        return *this;  
    }  
  
    tick operator++(int) {  
        auto self = *this;  
        ++*this;  
        return self;  
    }  
  
    // similar for decrement  
    tick &operator--();  
    tick operator-(int);  
  
private:  
    int cnt_;  
};
```

Overload: Member Access & Dereference

- Classes exhibit pointer-like behaviour when `->` and `*` are overloaded
- For `->` to work it must return a pointer to a class type or an object of a class type that defines its own `->` operator
- Useful for making “smart” pointers.

```
struct point { int x; int y; };

class point_ptr {
public:
    point_ptr(int x, int y)
        : ptr_{new point{x, y}} {}

    point &operator*() const {
        return *ptr;
    }

    point *operator->() const {
        return ptr_;
    }

    ~point_ptr() { delete ptr_; }

private:
    point *ptr_;
};
```

Overload: Type Conversions

- Define how a class type is converted into another type.
- Syntax for these operators look similar to constructors.
- Virtually always const-qualified.

```
struct centimeter { double cnt; };

struct inch {
    double cnt;

    operator centimeter() const {
        return centimeter{cnt * 2.45};
    }
};
```



Overload: Call Operator

- Make an instance of a class type callable
 - i.e., create *functors* (function objects)
- Can have many different overloads of operator() so long as they have different parameters.

```
struct comparator {  
    bool operator()(int l, int r) {  
        return l <= r;  
    }  
  
    bool operator()(char l, char r) {  
        return l <= r;  
    }  
};  
  
auto ints = std::vector<int>{3, 1, 2};  
auto chars = std::vector<char>{'c', 'a', 'b'};  
auto cmp = comparator{};  
  
std::sort(ints.begin(), ints.end(), cmp);  
std::sort(chars.begin(), chars.end(), cmp);
```



Operator Piggybacking

You'll notice that *many* operator overloads can be written in terms of other ones. The below table which operators are written in terms of which others.

Operator...	Can be written in terms of...
operator@ (where @ is one of the arithmetic or bitwise operators)	operator@=
operator!=	operator==
operator\$ (where \$ is one of <=, >, >=)	operator< and operator==
operator++(int)	operator++()
operator-(int)	operator--()
operator->	operator*

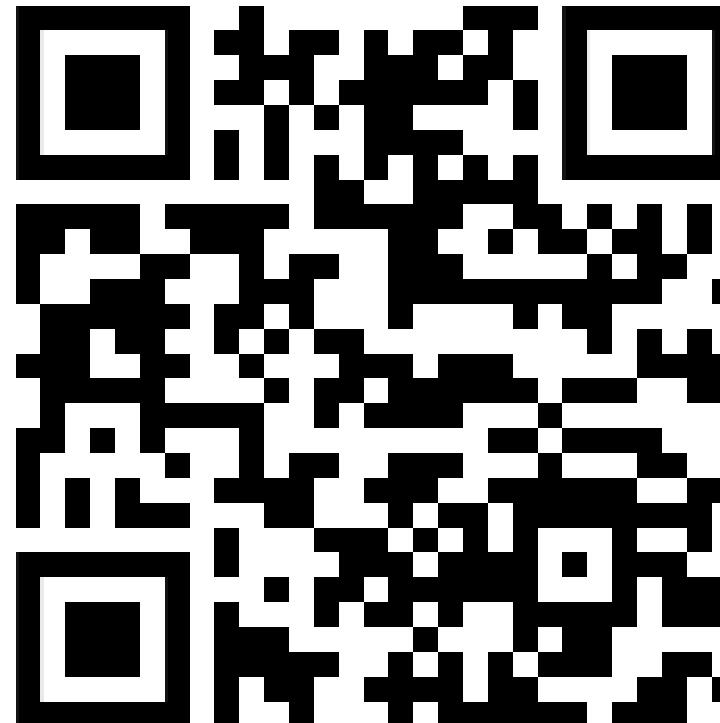


Miscellaneous Operators

- There are some operators that are rarely overloaded:
 - Allocation function (operator new / operator new[])
 - Deallocation function (operator delete / operator delete[])
 - The address-of operator (operator&)
 - The member-access-through-pointer operator (operator->*)
 - The comma operator (operator ,)
 - User-defined literals (since C++11) (operator "")
- There are also some operators that cannot be overloaded.
 - The ternary operator (operator?)
 - The member-access operator (operator.)
 - If this were possible, one could create “smart references”.



Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

4.2 – Custom Iterators



UNSW
SYDNEY

In this Lecture

Why?

- We sometimes need our custom types to be iterable.
- We must define that functionality ourselves

What?

- Iterator Revision
- How to Make a Custom Iterator
- Iterator Invalidation



Iterator Revision

- Iterator is an abstract notion of a pointer.
- Iterators are types that abstract container data as a linear sequence of objects.
- They allow containers and algorithms to interface generically.
 - Designers of algorithms need not care about how a container is implemented.
 - Designers of containers need not provide extensive operations.
- Iterators fall into distinct categories.
 - Output, Input, Forward, Bidirectional, Random-access, Contiguous

```
auto v = std::vector{1, 2, 3, 4, 5};  
const auto cv = v;  
  
// vector<int>'s non-const iterator  
++(*v.begin());  
  
// vector<int>'s const iterator  
*cv.begin();  
  
// vector<int>'s const iterator  
v.cbegin();
```

Custom Iterators

- A custom iterator is a class type that heavily uses operator overloading to provide the same syntactical operations as a pointer.
- A custom iterator must define certain traits for the compiler.
- Each category of iterator defines its set of operations.
 - [Base Iterator Requirements](#)
 - [Output Iterator Requirements](#)
 - [Input Iterator Requirements](#)
 - [Forward Iterator Requirements](#)
 - [Bidirectional Iterator Requirements](#)
 - [Random-Access Iterator Requirements](#)
 - [Contiguous Iterator Requirements](#)



Iterator Traits

- Every iterator has certain required type members.
 - Iterator category
 - Value type
 - Reference type
 - Pointer type
 - Not strictly required
 - Difference type
 - Used to count the number of elements between two iterators
- You must define these yourself in your custom iterators.

```
// iterator traits for an iterator
// modelling an int*
// <iterator> contains the category tags
#include <iterator>

class iter {
public:
    using iterator_category
        = std::contiguous_iterator_tag;

    using value_type = int;
    using reference_type = value_type&;
    using pointer_type = value_type*;
    // could also do pointer_type = void;
    // usually std::pointerdiff_t is sufficient
    using difference_type = std::pointerdiff_t;
};
```



Random-Access Iterator Interface

```
struct random_iter {  
    random_iter(); // must be default constructible.  
    random_iter(const random_iter &); // must be copy constructible.  
    random_iter& operator=(const random_iter &); // must be copy assignable.  
  
    reference operator*() const; // must be dereferenceable and return a reference.  
    pointer operator->() const; // only useful if this was an iterator to a class type  
  
    random_iter &operator++(); // must be pre-incrementable.  
    random_iter operator++(int); // must be post-incrementable.  
  
    random_iter &operator--(); // must be pre-decrementable.  
    random_iter operator--(int); // must be post-decrementable.  
  
    random_iter &operator+=(int n); // can progress n spots  
    random_iter &operator-=(int n); // can regress n spots  
  
    reference operator[](int); // get the nth element ahead from this position (setter version).  
    const reference operator[](int) const; // get the nth element ahead from this position (getter version).  
  
    friend random_iter operator+(random_iter, int n); // new iter n spots ahead  
    friend random_iter operator+(int n, random_iter); // new iter n spots ahead (reverse order)  
  
    friend random_iter operator-(random_iter, int n); // new iter n spots behind  
    friend difference_type operator-(random_iter, random_iter); // get the distance between two iterators  
  
    auto operator<=>(random_iter) const; // all six comparison functions are needed.  
};
```

From a Container to a Range

- A range is a container with certain member types and functions.
 - Particularly, a range can be used in a ranged for-loop.
- Member types:
 - iterator
 - const_iterator
 - Bidirectional and greater iterators also require:
 - reverse_iterator
 - const_reverse_iterator
- Member functions:
 - begin(), end()
 - cbegin(), cend()
 - Bidirectional and greater iterators also require:
 - rbegin(), rend()
 - crbegin(), crend()

```
class vector {  
    struct iter { /* implementation */ };  
public:  
    using iterator = iter;  
    using const_iterator = /* to be defined */;  
    using reverse_iterator = /* to be defined */;  
    using const_reverse_iterator = /* to be defined */;  
  
    iterator begin();  
    iterator end();  
    const_iterator begin() const;  
    const_iterator end() const;  
    const_iterator cbegin() const;  
    const_iterator cend() const;  
  
    reverse_iterator rbegin();  
    reverse_iterator rend();  
    const_reverse_iterator rbegin() const;  
    const_reverse_iterator rend() const;  
    const_reverse_iterator crbegin() const;  
    const_reverse_iterator crend() const;  
};
```

iterator & const_iterator

- The only practical difference between an iterator and a const_iterator is that the value_type of a const_iterator is const-qualified.
- This creates a potential problem of code duplication between const and non-const iterators.
- Solutions:
 - Accept the duplication? ✗
 - Give only one kind of iterator?
 - For some containers (like a set), only a const_iterator makes sense.
 - Use a template? ✓
 - We will cover templates later in the course.
 - Single-iterator types don't need templates

```
class vector {  
    template <typename ValueType>  
    struct iter {  
        // Instead of hardcoding a type  
        // directly, use the type parameter.  
        //  
        // Most other member types can be  
        // written in terms of value_type.  
        using value_type = ValueType;  
        Using reference = value_type&;  
        // more implementation...  
    };  
  
    using iterator = iter<int>;  
    using const_iterator = iter<const int>;  
  
    // more implementation...  
};
```

Automatic Reverse Iteration

- Reverse iterators can be created by using `std::reverse_iterator`.
- Requires a bidirectional iterator or greater.
- `rbegin()` stores `end()`, so `*rbegin` is actually `*(--end())`.

```
class vec {  
public:  
    using iterator = /* ... */;  
    using const_iterator = /* ... */;  
    using reverse_iterator =  
        std::reverse_iterator<iterator>;  
    using const_reverse_iterator =  
        std::reverse_iterator<const_iterator>;  
  
    iterator begin() { /* ... */ }  
    iterator end() { /* ... */ }  
  
    reverse_iterator rbegin() {  
        return reverse_iterator{end()};  
    }  
  
    reverse_iterator rend() {  
        return reverse_iterator{begin()};  
    }  
  
    // similar for other reverse iterator methods  
};
```



Iterator – Container Relationship

- Designers of containers usually provide the iterators also.
- The iterator must be at least publically default constructible, but usually has at least one private constructor.
- The container uses the private constructor to initialise the iterator to the start (if [cr]begin()) or end (if [cr]end()) of the range.
- This means that the container must be a friend of the iterator.
- Usually, the iterator class is defined as an *inner class* in the container.

int Stack Example: Container

Live Demo



int Stack Example: Iterator

Live Demo



Iterator Invalidation

- An iterator is an abstract notion of a **pointer**.
 - If the object a pointer points to moves, that pointer *dangles* and it is no longer valid to dereference.
- If the data an iterator references moves, it can dangle too.
- This is called **iterator invalidation**
 - No longer valid to dereference the iterator.
- Iterator invalidation is the consequence of (usually) adding or removing elements.
 - Element modification virtually never results in invalidation.

```
auto v = std::vector{1, 2, 3, 4, 5};
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it == 2) {
        v.push_back(2);
    }
} /* this for-loop copies all 2's. */

// the call to push_back may result in v's data
// being expanded and moved to a new location.
// if v.size() == v.capacity() when push_back()
// is called, it will expand.

// if it did not expand, only variables holding
// the old v.end() are invalidated.

// if it did expand, all iterator variables are
// invalidated and cannot be used.
```

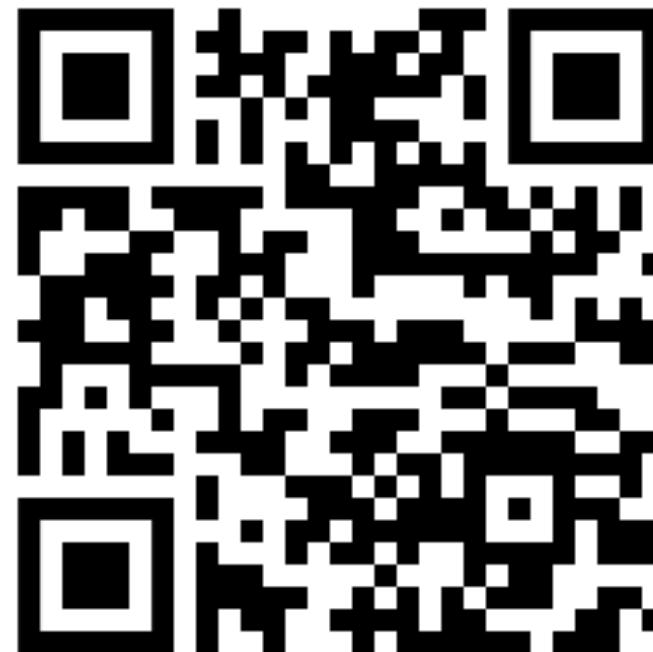


Iterator Invalidation in the STL

- Iterators from array-backed containers (`vector`, `unordered_map`, `unordered_set`, etc.) are invalidated when the array needs to grow or shrink.
 - For `std::vector`, this happens most often through `push_back()`.
 - For the unordered containers, this happens most when a rehash of elements is needed.
- Iterators from linked data structures (`list`, `map`, `set`, etc.) are only invalidated when elements are removed.
 - Size changes result in internal pointers being adjusted, but these don't affect the iterators.



Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

5.1 - Exceptions



UNSW
SYDNEY

In this Lecture

Why?

- Sometimes our programs need to deal with unexpected runtime errors and handle them gracefully.

What?

- Throwing and catching exceptions
- Exception safety levels
- Exception performance



A Motivating Example

- What does the code produce?
- What can go wrong?
 - Call to malloc() in vector could fail?
 - Conversion of string to int could fail when reading in user input?
 - User inputted an index too big?

```
#include <vector>
#include <iostream>

int main() {
    std::cout << "Enter -1 to quit\n";
    std::vector<int> items = {97, 84, 72, 65};
    std::cout << "Enter an index: ";
    for (int ix; std::cin >> ix && ix != -1;) {
        std::cout << items[ix] << "\n";
        std::cout << "Enter an index: ";
    }
}
```

A (Correct) Motivating Example

- What does the code produce?
- What can go wrong?
 - Call to malloc() in vector could fail
 - unrecoverable error.
 - Conversion of string to int could fail when reading in user input
 - build that into the design of reading from std::cin.
 - User inputted an index too big
 - use dynamic exceptions!

```
#include <vector>
#include <iostream>

int main() {
    std::cout << "Enter -1 to quit\n";
    std::vector<int> items = {97, 84, 72, 65};
    std::cout << "Enter an index: ";
    for (int ix; std::cin >> ix && ix != -1;) {
        try {
            std::cout << items.at(ix) << "\n";
        } catch (std::out_of_range &e) {
            std::cout << "Index out of bounds\n";
        } catch(...) {
            std::cout << "Something went wrong!\n";
        }
        std::cout << "Enter an index: ";
    }
}
```



C++ Exceptions

- **Exceptions** are for exceptional circumstances.
 - Problems can happen through execution (things not going to plan A!)
- **Exception Handling:**
 - Run-time (dynamic) mechanism
 - Code downstream detects anomalies and throws an appropriate exception
 - Upstream (sometimes unrelated) code catches the exception, handles it, and potentially rethrows it.
- **Why?**
 - Allows us to gracefully and programmatically deal with anomalies, as opposed to our program crashing.



Conceptual Structure

- Exceptions are just data that is “thrown” up the stack when an error is detected.
 - Usually are instances of classes that extend `std::exception`.
 - Limited type conversions exist for exceptions:
 - From non-const to const (but not the reverse!).
 - User-defined conversions.
- Code that can throw is placed in a `try` block.
- Code to handle a thrown exception is placed in a `catch` block.
- Stack unwinds until an appropriate `catch` block is found.

```
// for standard exception definitions
#include <exception>

try {
    // Code that can throw
} catch (const std::out_of_range &) {
    // Code that can handle an out-of-range
    // error (maybe from misuse of a vector?)
} catch (const std::exception &) {
    // Code that can catch all exceptions
    // that derive from std::exception
    // (excluding std::out_of_range)
} catch (...) {
    // Code that can catch anything
}
```

Catching the Right Way

- **Throw by value, catch by const reference**
- Ways to catch exceptions:
 - By pointer (no!)
 - By value (no!)
 - By reference (yes !!)
- References are preferred because:
 - more efficient, less copying (exploring today)
 - no slicing problem (related to polymorphism, exploring later)
- [Extra reading for those interested](#)



Rethrowing

- When an exception is caught, by default that catch block will be the only part of the code with access to the exception.
- To give other catch blocks upstream access to the exception, can throw again.

```
try {  
    try {  
        std::cout << "oops" << std::endl;  
        throw 6771;  
    } catch (const int &x) {  
        std::cout << "exception detected."  
            << "rethrowing\n";  
        throw x;  
    }  
} catch (const int &i) {  
    std::cout << "i: " << i << std::endl;  
}  
//changing exception type
```



Multiple catch Blocks

- A single try block can have more than one catch.
- Each catch block argument is matched against the current exception until an appropriate type is found.
- Should order the catch blocks from most to least specific exception for maximum performance.
- `catch(...)` should almost always be last.

```
#include <iostream>
#include <vector>

int main() {
    auto items = std::vector<int>{};
    try {
        items.resize(items.max_size() + 1);
    } catch (const std::bad_alloc &e) {
        std::cout << "Out of bounds.\n";
    } catch (const std::exception&) {
        std::cout << "General exception.\n";
    } catch (...) {
        std::cout << "Even more general.\n";
    }
}
```



noexcept: asking if the exception can't throw

- Specifies whether a function could potentially throw.
- Querying at compile time—never to emit an exception.
- It doesn't actually prevent a function from throwing an exception.
- If a noexcept function throws, `std::terminate` is called.
- Compiler can sometimes make optimisations if it knows a function is noexcept.
- Improve algorithmic performance in generic code. Move-Big(0) faster
- Precisely provide info, what we need at compile time.

```
class S1 {
public:
    // may throw
    int foo() const;
};

class S2 {
public:
    // does not throw
    int foo() const noexcept;
};

void foo() noexcept {
    // cannot throw either.
}
```

Stack Unwinding

- **Stack Unwinding** is the automatic process of popping stack frames until an appropriate handler for an exception is found.
- Once a found catch block successfully exits without rethrowing, stack unwinding is complete.
- If it catches by value, its formal parameter is initialized by copying the exception object. If it catches by reference, the parameter is initialized to refer to the exception object, then unwinding
- If another exception is thrown during stack unwinding before a catch block is found, `std::terminate` is called.



Exceptions & Destructors

- Compiler ensures any destructors are called during stack unwinding.
- All exceptions that occur inside a destructor **must** be handled inside the destructor.
- Therefore, all destructors are implicitly `noexcept` .



Exceptions & Constructors

- What happens if an exception is thrown halfway through a constructor?
- The C++ Standard: "An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed subobjects"
- A destructor is not called for an object that was partially constructed.
- ...except for an exception thrown in a delegating constructor (why?).
- Common problems:
 - Code that catches an exception thrown in a constructor assumes the object is fully constructed and uses it.
 - A failure partway through an object's construction leads to its destructor running against a partially-constructed object.
 - Resources may be leaked.

```
// spot the bug

class unsafe_class {
public:
    unsafe_class(int a)
        : a_{new int{a}},
        : i_{a == 7 ? a : throw "uh oh"}
    {}

    ~unsafe_class() {
        delete a_;
    }

private:
    int *a_;
    int i_;
};

int main() {
    auto a = unsafe_class(6771);
}
```

Exception: when/when not ?

- Rare error
- Exceptional case that can not be dealt locally (I/O error)
- File not found
- Can't find key in map
- Operators and constructor where no other mechanism works
- For errors that occurs frequently.
- Function that are expected to fail.

```
auto s_int(std::string const& st)->std::optional<int>
auto s_int(std::string const& st)->boost::outcome<int>
auto s_int(std::string const& st)->std::expected<int>
```
- Have to guarantee response time even in error
- Things that should never happen
 - Use after free
 - Out of range
 - Dereferencing nullptr



std::expected/unexpected

Any given time, it either holds an expected value of type T, or an unexpected value of type E.

Directly allocated within the storage occupied by the expected object. No dynamic memory allocation takes place.

Represents an unexpected value stored in std::expected.

std::expected has constructors with std::unexpected as a single argument, which creates an expected object that contains an unexpected value.

```
std::expected<double, int> ex = std::unexpected(3);
if (!ex) std::cout << "ex contains an error value\n";
if (ex == std::unexpected(3)) std::cout << "The error value is equal to 3\n";
```



Exception Safety Levels

- This is not specific to C++.
- It is about writing exception-safe code.
 - Keeping the program in consistent state even after an exception is thrown.
- Operations performed have various levels of safety:
 - No-throw (failure transparency).
 - Strong exception safety (commit-or-rollback).
 - Weak exception safety (no-leak).
 - No exception safety.



No-throw Guarantee

- Also known as failure transparency.
- Operations are guaranteed to succeed, even in exceptional circumstances.
 - Exceptions may occur but are handled internally.
 - No exceptions are visible to the client.
- Implemented in C++ with `noexcept`.
- Examples:
 - Closing a file.
 - Freeing memory.
 - Anything done in destructors.
 - Creating a trivial object on the stack (all types from C are trivial).



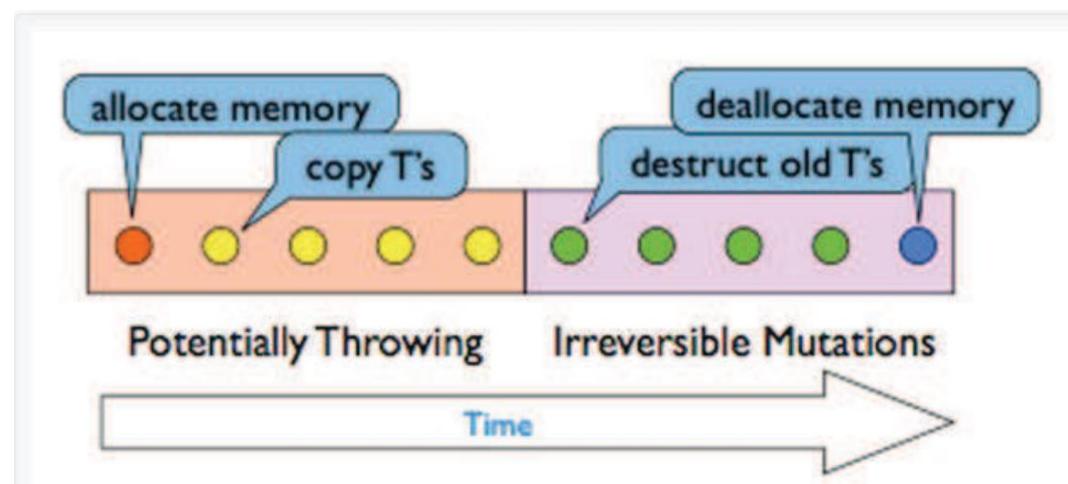
Strong Exception Safety

- Also known as "commit or rollback" semantics.
- Operations can fail, but failed operations are guaranteed to have no visible effects.
- Probably the most common level of exception safety for types in C++.
- All your copy-constructors should generally follow these semantics.
- Similar for copy-assignment.
 - Can be difficult when manually writing copy-assignment



Implementing Strong Safety

- To achieve strong exception safety, you need to:
 - First perform any operations that may throw, but don't do anything irreversible.
 - Then perform any operations that are irreversible, but don't throw.
 - So-called "copy & swap" idiom.



```
strong& operator=(strong const& other) {  
    strong temp(other); // copy  
    temp.swap(*this); // ...and swap  
    return *this;  
}
```

Strong guarantee can be costly i.e. if the `Strong` object in the example allocates large amounts of memory. Instead of reusing the already allocated memory, the temporary has to allocate new memory just to release the old one after the swap.



Basic Exception Safety

- This is known as the no-leak guarantee.
 - we can be sure that our objects class invariants are not violated. Nothing more, nothing less.
- Change in status of program before exception thrown.
- Partial execution of failed operations can cause side effects, but:
 - All invariants must be preserved.
 - No resources are leaked.
 - No data corruption.
- Any stored data will contain valid values, even if different from before the exception was thrown.
 - A "valid, but unspecified state".



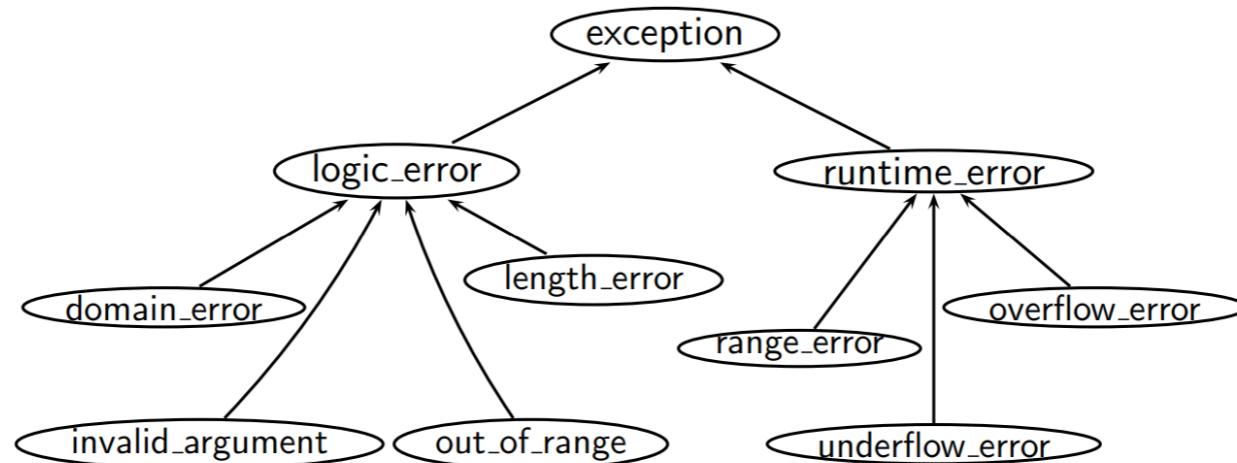
No Exception Safety

- No guarantees.
- Don't write C++ with no exception safety.
- Very hard to debug when things go wrong.
- Very easy to fix:
 - Be mindful about object lifetimes.
 - Where possible, use the standard library.
 - This gives you basic exception safety for free.



Standard Exceptions

- Standard Library defines its own exception hierarchy.
- If a standard library type throws, it throws one of these exceptions.
- Custom exception objects should derive from the most appropriate standard exception.



Exception Performance

- Exceptions are for exceptional circumstances.
 - They should **not** be used for control flow if a better alternative exists.
 - E.g., instead of throwing an exception, could we return a `std::optional` instead?
- Performance-critical code largely does not use exceptions.
 - A single exception can use 1000 or more CPU cycles before completion.
- Exceptions are best for cold code paths.
- [\(Optional\) Further reading](#) that could improve exception performance.



Feedback



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

5.2 – Resource Management



UNSW
SYDNEY

In this Lecture

Why?

- Performance & Control
 - with great power comes great responsibility.
- Leak freedom in C++.
- C++ is not garbage collected.
- While we have ignored heap resources (`malloc/free`) to date, they are a critical part of many libraries, and we need to understand best practices around usage.

What?

- Resource Management
- RAII
- Smart Pointers



UNSW
SYDNEY

Revision: Objects

- What is an object in C++?
 - An object is a region of memory associated with a type.
 - Unlike some other languages (Java), basic types such as `int` and `bool` are objects.
- For the most part, C++ objects are designed to be intuitive to use.
- Special things can we do with objects:
 - Create them;
 - Destroy them;
 - Copy them; and
 - Move them.



Long Lifetimes

- There are 3 ways you can try and make an object in C++ have a lifetime that outlives the scope it was defined in:
 - Returning it out of a function via copy (can have limitations).
 - Returning it out of a function via references (leads to dangling references).
 - Returning it out of a function as a heap resource (explicit memory management).

```
struct point2i { int x; int y; }

// This function returns a new object,
// Not a reference to the object
point2i make_point_copy(int x, int y) {
    point2i p = {x, y};
    return p;
}

// Returns a reference to a stack-local variable.
// Program has undefined behaviour if used.
point2i &make_point_dangling(int x, int y) {
    point2i p = {x, y};
    return p;
}

// Returns a reference to a heap-allocated variable.
// Now we must manage the memory explicitly.
point2i &make_point_heap(int x, int y) {
    point2i *ptr = new point2i{x, y};
    return *ptr;
}
```



Long Lifetimes & References

- We need to be very careful when returning references.
- *The object must always outlive the reference.*
- Using dangling references is undefined behaviour!
- **Moral of the story:** Do not return references to stack-local variables.
- Unless we copy, for objects we create INSIDE a function, we must heap-allocate them!

```
auto okay(int &i) -> int& {  
    return i;  
}  
  
// why is this ok?  
// answer: lifetime extension  
auto okay(int &i) -> const int& {  
    return i;  
}  
  
auto not_okay(int i) -> int& {  
    return i;  
}  
  
auto definitely_not_okay() -> int& {  
    auto i = 0;  
    return i;  
}
```

new & delete

- Non-global objects are either stored on the *stack or heap*.
- In general, you've created most objects on the stack (without realising!)
- We can allocate heap objects via new and free them via delete.
 - new and delete call the constructors/destructors of what they are creating.

```
#include <cstdlib>
#include <iostream>
#include <vector>

int main() {
    int *a = new int{4};
    auto *b = new std::vector<int>{1,2,3};

    std::cout << *a << "\n";
    std::cout << (*b)[0] << "\n";

    delete a; // frees a
    delete b; // frees b

    return EXIT_SUCCESS;
}
```

A Heap of Memory

- Why do we need heap resources?
 - Heap objects can outlive the scopes they were created in.
 - More useful in contexts where we need explicit control of ongoing memory size e.g. for vector.
 - Stack has limited space on it for storage, heap is much larger.
 - No matter how much we try, it is very difficult to use all of dynamically allocated memory.
 - Primary exception is on embedded systems.

```
#include <iostream>
#include <vector>

int *make_int(int i) {
    int *a = new int{i};
    return a;
}

int main() {
    int *a = make_int(6771);

    // a was defined in a scope that
    // no longer exists
    std::cout << *a << "\n";

    delete a;
}
```

Heap Allocation is Not Free

- Non-trivial programs almost always use the heap in some way.
 - All of our examples using `std::vector` have been using the heap secretly as well.
- Heap allocation is expensive compared to stack allocation.
 - Should be avoided in performance-critical code.
- C++ has value semantics.
 - Is it possible to avoid the cost associated with deep-copying a heap allocated object?
- Answer: Move Semantics!



Move Semantics (C++11 onwards)

- Move semantics are a way for class-types to “steal” the data of a temporary value and reuse it.
- Makes use of a language feature called **rvalue references**.
- Let’s review lvalues and rvalues.
 - Also called “value categories”.

```
#include <vector>

std::vector<int> make_vec() {
    return std::vector<int>{1, 2, 3, 4, 5};
}

int main() {
    // here, v is default initialised.
    std::vector<int> v;

    // The vector from make_vec() is a temporary.
    // Rather than make v allocate a vector that
    // can fit the contents of make_vec(), why not
    // just steal make_vec()'s data directly?
    // It is going out of scope, so nothing will be
    // affected by this.
    v = make_vec(); // make_vec() is “moved” into v.
}
```



lvalue vs rvalue

- **lvalue:** An expression that is an object reference.
 - E.g. named variables.
 - You can always take the address of an lvalue.
- **rvalue:** Expression that is not an lvalue
 - E.g.. Object literals, return result of functions.
 - rvalues are temporary and short lived, while lvalues live a longer life since they exist as variables.

```
int &f();  
  
int main() {  
    // 5 is rvalue, i is lvalue  
    int i = 5;  
  
    // j is lvalue, i is lvalue  
    int j = i;  
  
    // 4 + i produces rvalue then stored in lvalue k  
    int k = 4 + i;  
  
    // error: cannot assign to rvalues.  
    6 = k;  
  
    // taking address of an lvalue  
    int* y = &k;  
  
    // error: cannot take address of an rvalue.  
    int* y = &6771;  
  
    // OK: f() returns an lvalue reference  
    f() = 3;  
}
```

Ivalue References

- There are multiple types of references:
 - Ivalue references look like `T&`.
 - Ivalue references to const look like `const T&`.
- Once the Ivalue reference goes out of scope, the original Ivalue is still usable.
- Constant Ivalue references exhibit **lifetime extension**.
 - Any temporaries bound to a `const T&` act like an Ivalue.
 - The temporary gets stored in memory and is referred to via the reference

```
// regular lvalue
int y = 10;

// OK: binding lvalue to an lvalue reference.
int &yref = y;

// OK: binding lvalue to a const lvalue
// reference.
const int &cref = y;

// !!: cannot bind const lvalue to a non-const
// ref
int &ref = static_cast<const int>(y);

// !!: rvalues only bind to const lvalue
// references
const int &extended = 10;
```

rvalue References

- rvalue references look like T&&.
- rvalue references extend the lifespan of the temporary object to which they are assigned.
- Non-const rvalue references allow you to modify the rvalue.
- An rvalue reference formal parameter means that the value was disposable from the caller of the function.
 - If the callee modified value, who would notice or care?
 - The caller has promised that it won't be used anymore
 - An rvalue reference parameter is an lvalue inside the function.

```
#include <iostream>

// Declaring an rvalue reference
int &&rref = 20;

void inner(std::string &&value) {
    value[0] = 'H';
    std::cout << value << '\n';
}

void outer(std::string &&value) {
    inner(value); // This call fails? Why?
    std::cout << value << '\n';
}

int main() {
    outer("hello"); // This call succeeds.
    auto s = std::string("hello");

    // This call fails because s is an lvalue.
    inner(s);
}
```

std::move

Uses of rvalue references:

- They are used in working with the move constructor and move assignment special member functions.
- cannot bind non-const lvalue reference of type '**int&**' to an rvalue of type '**int**'.
- cannot bind rvalue references of type '**int&&**' to lvalue of type '**int**'.
- A library function that converts an lvalue to an rvalue so that a "move constructor" (similar to copy constructor) can use it.
- This says "I don't care about this anymore".
- All this does is allow the compiler to use rvalue reference overloads.

```
// std::move looks something like this.  
T&& move(T& value) {  
    return static_cast<T&&>(value);  
}  
  
void inner(std::string&& value) {  
    value[0] = 'H';  
    std::cout << value << '\n';  
}  
  
void outer(std::string &&value) {  
    // this now works!  
    inner(std::move(value));  
  
    // Value is now in a valid but unspecified state.  
    // Although this isn't an error, this is bad code.  
    // Only access moved variables to remake them.  
    std::cout << value << '\n';  
}  
  
int main() {  
    outer("hello"); // This works fine.  
    auto s = std::string("hello");  
    inner(s); // This fails because i is an lvalue.  
}
```

RAII: Resource Acquisition is Initialisation

- A resource is anything that is scarce and must be managed.
 - E.g., Memory, locks, file descriptors, time.
- RAII is a concept where we encapsulate resources inside objects.
 - Acquire the resource in the constructor.
 - Release the resource in the destructor.
 - Deep-copy the resource if it makes sense.
 - Transfer ownership if it makes sense.
 - Resource is always released at a known point in the program, which you can control.
- Every resource should be owned by either:
 - Another resource (e.g. smart pointer, data member).
 - Named resource on the stack.
 - A nameless temporary variable.



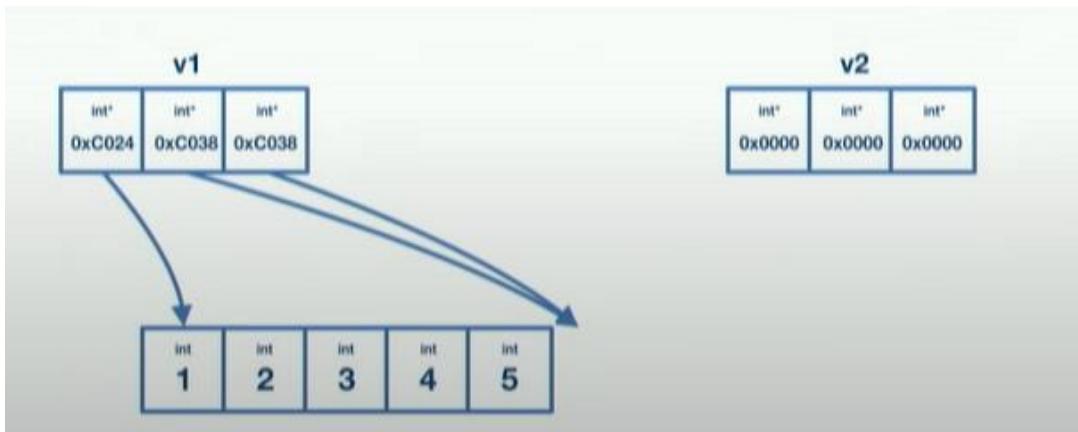
RAll-aware Classes

- When writing a class, if we can't default all of our operators (preferred), we should consider the **Rule of 5**:
 1. Destructor
 2. Move assignment
 3. Move constructor
 4. Copy constructor
 5. Copy assignment
- The presence or absence of these 5 operations is critical in managing resources.
- If you define one of these special member functions, you should explicitly define or default all of them.



Implementing an RAII Class

- Let's try implementing our own version of `std::vector`.
- It's going to have to manage some heap memory, so it should look something like this.



```
class vec {  
public:  
    vec(int n)  
        : ptr_{new double[n]}, size_{n}, cap_{n} {}  
  
    vec(const vec &other);  
    vec(vec &&other);  
  
    vec &operator=(const vec &other);  
    vec &operator=(vec &&other) noexcept;  
  
    ~vec();  
  
    // other implementation...  
private:  
    double *ptr_;  
    int size_;  
    int cap_;  
};
```

RAII Class: Destructor

- What happens when `v` goes out of scope?
 - Destructors are called on each member...
 - But destructing a pointer type does nothing!
- As it stands, this will result in a memory leak.
 - How do we fix?
 - Use the destructor!

```
class vec {  
public:  
    // other implementation...  
    ~vec() noexcept /* optional noexcept specifier */ {  
        delete[] ptr_;  
    }  
private:  
    double *ptr_;  
    int size_;  
    int cap_;  
};  
int main() {  
    auto v = vec{4};  
} // went out of scope...problem?
```



RAII Class: Move Constructor

- Move constructor should be noexcept.
- Can use std::exchange to exchange the important data members from the temporary.
- Often not much more complex than this.

```
class vec {  
public:  
    // other implementation...  
    vec(vec &&other) noexcept  
        : data_{std::exchange(other.data_, nullptr)},  
          size_{std::exchange(other.size_, 0)},  
          cap_{std::exchange(other.cap_, 0)} {}  
  
private:  
    double *ptr_;  
    int size_;  
    int cap_;  
};
```



RAII Class: Move Assignment

- Like the Move Constructor, but the destination has already been constructed.
- The easiest way to write a move assignment is generally to do member-wise swaps, then clean up the original object.
- Doing so may mean some redundant code, but it means you don't need to deal with mixed state between objects.

```
class vec {  
vec &operator=(vec &&other) noexcept {  
    if (this != &other) {  
        std::swap(data_, other.data_);  
        std::swap(size_, other.size_);  
        std::swap(cap_, other.cap_);  
  
        delete[] other.data_;  
        other.data_ = nullptr;  
        other.size_ = 0;  
        other.capacity = 0;  
    }  
}  
// other implementation...  
};
```



Moving Objects

- Always declare your moves as `noexcept` (why?)
 - Failing to do so *might* make your code slower.
- Unless otherwise specified, objects that have been moved from are in a valid but unspecified state.
- Moving is an optimisation on copying
 - The only difference is that when moving, the moved-from object is mutable.
 - Not all types can take advantage of this:
 - If moving an `int`, mutating the moved-from `int` is extra work
 - If moving a `vector`, mutating the moved-from `vector` potentially saves a lot of work.
- Moved from objects must be placed in a valid state.
 - Moved-from containers usually contain the default-constructed value.
 - Moved-from types that are cheap to copy are usually unmodified.
 - Although this is the only requirement, individual types may add their own constraints.
- Compiler-generated move constructor / assignment performs member-wise moves.

RAII Class: Copy Constructor

- What does the default synthesised copy constructor do?
 - Member-wise copy.
- What are the consequences?
 - Any modification to our vec instance will also change the original.
 - Likely lead to a double-free.
- How can we fix this?
 - Custom copy constructor that does a deep copy!
 - Copy constructors are **rarely noexcept** (why?).

```
class vec {  
public:  
    // other implementation...  
    vec(const vec &o)  
        : data_{new double[o.size_]},  
          size_{o.size_},  
          cap_{o.cap_} {  
        std::copy(o.data_, o.data_ + o.size_, data_);  
    }  
  
private:  
    double *ptr_;  
    int size_;  
    int cap_;  
};
```

RAII Class: Copy Assignment

- Assignment is the same as construction, except there is already a constructed object in your destination.
- You need to clean up the destination first.
- The copy-and-swap idiom makes this trivial.

```
class vec {  
public:  
    // other implementation...  
    vec& operator=(const vec &other) {  
        vec(other).swap(*this);  
        return *this;  
    }  
private:  
    void swap(vec &other) {  
        std::swap(data_, other.data_);  
        std::swap(size_, other.size_);  
        std::swap(capacity_, other.capacity_);  
    }  
};
```



Explicitly Deleted Copy & Moves

- We may not want a type to be copyable / moveable.
 - Non-copyable types implement “unique ownership” semantics.
 - Only this object owns this resource but can transfer ownership.
 - Non-moveable types are rare.
 - Best example is [std::scoped_lock](#)
 - If we want to delete them, we can declare
`fn() = delete;`

```
struct T {  
    T(const T&) = delete;  
    T(T&&) = delete;  
  
    T& operator=(const T&) = delete;  
    T& operator=(T&&) = delete;  
};
```



Implicitly Deleted Copies & Moves

- Under certain conditions, the compiler will not generate the copy and move member functions.
- The implicitly defined copy constructor calls the copy constructor member-wise.
 - If one of its members doesn't have a copy constructor, the compiler can't generate one for you.
 - Same applies for copy assignment, move constructor, and move assignment.
- Under certain conditions, the compiler will not automatically generate copy / move assignment / constructors.
 - If you have manually defined a destructor, the move constructor isn't generated.
 - If you have manually defined a move constructor, the copy constructor isn't generated.
- If you define one of the Five, you should explicitly delete, default, or define all of the Five.
 - If the default behaviour isn't sufficient for one of them, it likely isn't sufficient for others
 - Explicitly doing this tells the reader of your code that you have carefully considered this
 - This also means you don't need to remember all of the rules about "if I write X, then is Y generated?"



RAII Case Study: Smart Pointers

- Introduced in C++11.
- Ways of wrapping unnamed (i.e., raw pointer) heap objects in named stack objects so that object lifetimes can be managed much easier.
- Supports automatic memory management
 - allocate/deallocate according to RAII.
- Usually two ways of approaching problems:
 - unique_ptr + raw pointers
 - shared_ptr + weak_ptr

Type	Implied Ownership
<code>T*</code>	None
<code>std::unique_ptr<T></code>	Sole ownership
<code>std::shared_ptr<T></code>	Shared ownership
<code>std::weak_ptr<T></code>	None



std::unique_ptr

- When the unique pointer is destructed, the underlying object is too.
- Slightly more overhead than raw pointers.
- Can be parameterized with a custom deleter
 - default deleter uses `delete`.
- Array version exists:
 - Use `std::unique_ptr<T[]>` instead.

```
#include <memory>
#include <iostream>

int main() {
    auto up1 = std::unique_ptr<int>{new int{42}};

    // error: no copy constructor
    auto up2 = up1;

    std::unique_ptr<int> up3;
    // no copy assignment
    up3 = up2;

    up3.reset(up1.release()); // OK
    auto up4 = std::move(up3); // OK

    std::cout << up4.get() << "\n";
    std::cout << *up4 << "\n";
    std::cout << *up1 << "\n";
} // dynamically allocated int freed here.
```



Raw (Observer) Pointers

- Used to “observe” a `unique_ptr`.
- This is an appropriate use of raw pointers in C++.
- Once the original pointer is destructed, you must ensure you don't access the raw pointers (no checks exist).
- These observers do not have ownership of the pointer.
- Also note the use of `nullptr` in C++ instead of NULL .

```
#include <memory>
#include <iostream>

int main() {
    auto up1 = std::unique_ptr<int>(new int{5});
    auto op1 = up1.get();
    *op1 = 6;

    std::cout << *op1 << "\n"; // prints 6
    std::cout << *up1 << "\n"; // also prints 6

    // free the managed int
    up1.reset(nullptr);

    // undefined behaviour
    std::cout << *op1 << "\n";
}
```



std::shared_ptr

- Several objects share ownership of the underlying resource.
- Uses reference counting to avoid premature freeing.
- When a shared pointer is destructed, if it is the only shared pointer left pointing at the object, then the object is destroyed.
- Observers are `std::weak_ptr`s instead of raw pointers.
- Like `unique_ptr`, an array version is also available.

```
#include <iostream>
#include <memory>

auto main() -> int {
    auto x = std::make_shared<int>(5);

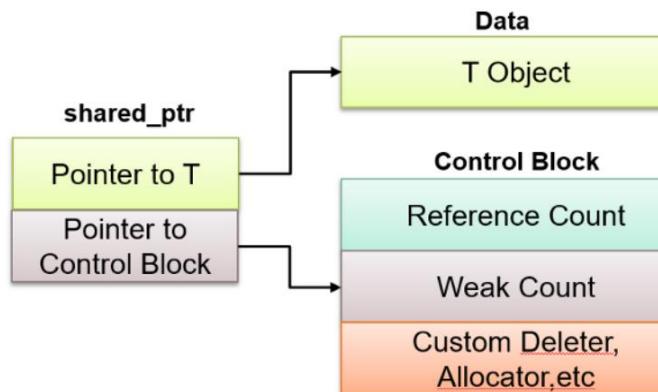
    std::cout << "use count: " << x.use_count() << "\n";
    std::cout << "value: " << *x << "\n";

    x.reset(); // Memory still exists, due to y.
    std::cout << "use count: " << y.use_count() << "\n";
    std::cout << "value: " << *y << "\n";

    // Deletes the memory, since no one else owns the memory
    y.reset();

    std::cout << "use count: " << x.use_count() << "\n";
    std::cout << "value: " << *y << "\n";
}
```

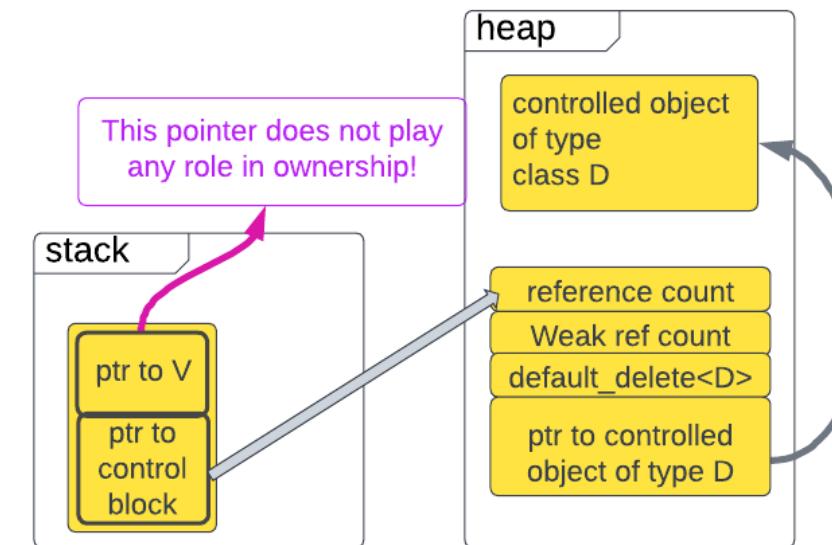
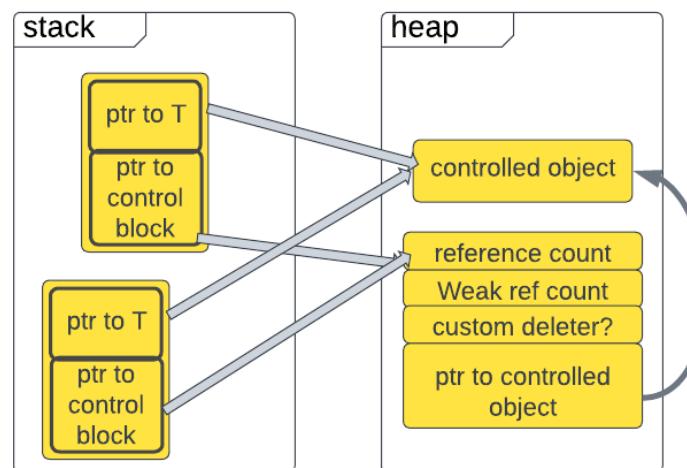
std::shared_ptr Control Block



`shared_ptr`, unlike `unique_ptr`, places a layer of indirection between the physical heap-allocated object and the notion of ownership.

`shared_ptr` instances are essentially participating in ref-counted ownership of the control block.

The control block itself is the sole arbiter of what it means to “delete the controlled object.”



std::weak_ptr

- Weak pointers are used with shared pointers when:
 - You don't want to add to the reference count.
 - You want to be able to check if the underlying data is still valid before using it.
 - Break a circular dependency.
- Must be converted to a shared_ptr with the lock() method before it can be used.

```
#include <iostream>
#include <memory>

auto main() -> int {
    auto x = std::make_shared<int>(1);
    // x owns the memory
    auto wp = std::weak_ptr<int>(x);
    // now y is a shared_ptr and can be used.
    auto y = wp.lock();

    if (y != nullptr) {
        // x and y own the memory.
        // Do something with y.
        std::cout << "Attempt 1: " << *y << '\n';
    }
}
```

When to Use Which Pointer

Unique Pointer vs Shared Pointer

- You almost always want a `unique_ptr` over a `shared_ptr`
- Use a `shared_ptr` if either:

- An object has multiple owners, and you don't know which one will stay around the longest; or
- You need temporary ownership
 - This is very rare.
 - Outside scope of this course.

“Leak freedom in C++” poster

Strategy	Natural examples	Cost	Rough frequency
1. Prefer scoped lifetime by default (locals, members)	Local and member objects – directly owned	Zero: Tied directly to another lifetime	O(80%) of objects
2. Else prefer <code>make_unique</code> & <code>unique_ptr</code> or a container, if the object must have its own lifetime (i.e., heap) and ownership can be unique w/o owning cycles	Implementations of trees, lists	Same as new/delete & malloc/free Automates simple heap use in a library	O(20%) of objects
3. Else prefer <code>make_shared</code> & <code>shared_ptr</code> , if the object must have its own lifetime (i.e., heap) and shared ownership w/o owning cycles	Node-based DAGs, incl. trees that share out references	Same as manual reference counting (RC) Automates shared object use in a library	

Don't use owning raw *'s == don't use explicit delete

Don't create ownership cycles across modules by owning “upward” (violates layering)
Use `weak_ptr` to break cycles

Function Signature	Ownership Semantic
<code>func(value)</code>	<ul style="list-style-type: none">▪ Is an independent owner of the resource▪ Deletes the resource automatically at the end of <code>func</code>
<code>func(pointer*)</code>	<ul style="list-style-type: none">▪ Borrows the resource▪ The resource could be empty▪ Must not delete the resource
<code>func(reference&)</code>	<ul style="list-style-type: none">▪ Borrows the resource▪ The resource could not be empty▪ Must not delete the resource
<code>func(std::unique_ptr)</code>	<ul style="list-style-type: none">▪ Is an independent owner of the resource▪ Deletes the resource automatically at the end of <code>func</code>
<code>func(shared_ptr)</code>	<ul style="list-style-type: none">▪ Is a shared owner of the resource▪ May delete the resource at the end of <code>func</code>



Smart Pointer Factory Functions

- `make_shared()` and `make_unique()` wrap raw `new`, just as `~shared_ptr()` and `~unique_ptr()` wrap raw `delete`.
- Pass the arguments you would have passed to the underlying object's constructor to these functions.
- Never touch raw pointers with hands, and then never need to worry about leaking them.
- `make_unique()` prevents the *unspecified-evaluation-order* leak bug triggered by expressions like:
 - `foo(unique_ptr<T>(new T()), unique_ptr<U>(new U()));`
 - Above, if either the constructor of `T` or `U` throw and the other object has already been allocated, the destructor of `unique_ptr` won't be called.
- `make_shared()` is faster than using `shared_ptr`'s constructors.
 - Able to allocate the managed object and control block in a single allocation rather than two.



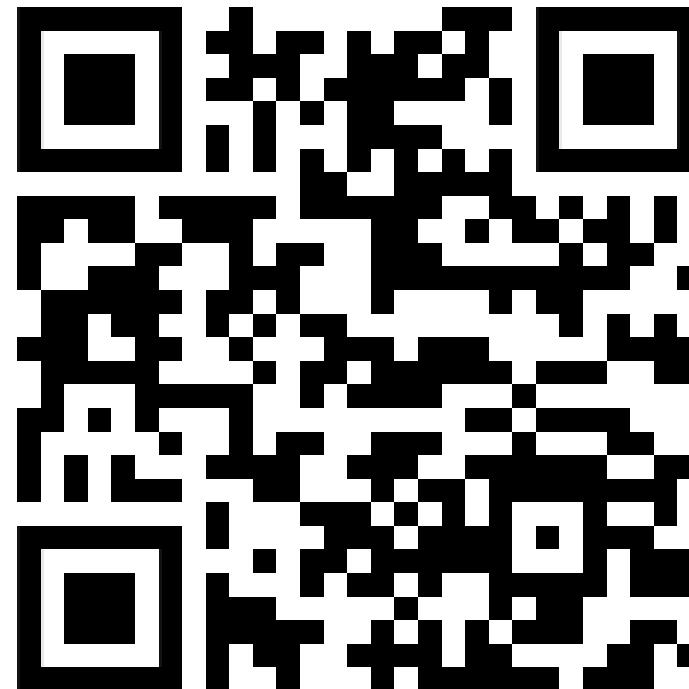
Resource Safety

To ensure resource safety in C++, we always attach the lifetime of one object to that of something else.

- **Named objects:**
 - A variable in a function is tied to its scope.
 - A data member is tied to the lifetime of the class instance.
 - An element in a `std::vector` is tied to the lifetime of the vector.
- **Unnamed objects:**
 - A heap object should be tied to the lifetime of whatever object created it.
 - Ideally this should be a smart pointer.
 - Examples of *bad programming practice*:
 - An owning raw pointer is tied to nothing.
 - A heap-allocated C-style array is tied to nothing.



Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

7.1 Dynamic Polymorphism



UNSW
SYDNEY

Key Concepts

- Inheritance
 - To be able to create new classes by inheriting from existing classes.
 - To understand how inheritance can promote software reusability.
 - To understand the notions of base classes and derived classes.
- Polymorphism
 - Dynamic: determine which method to call at run-time
 - Static: determine which method to call at compile-time
 - Considerations of Dynamic Polymorphism



Use Cases for Dynamic Polymorphism

- Dynamic polymorphism allows for an “open” type system.
 - Types can be extended with new types.
 - Types can be customised beyond what the original author imagined.
- Natural fit when types have an “is-a” relationship.
- Can reduce code duplication for extremely similar types.

```
#include <iostream>
#include <vector>

struct mammal {
    virtual auto speak() const -> void;
};

struct dog : mammal {
    auto speak() const -> void override {
        std::cout << "wan" << std::endl;
    }
};

struct cat : mammal {
    auto speak() const -> void override {
        std::cout << "nyaa" << std::endl;
    }
};

int main() {
    dog d;
    cat c;
    std::vector<mammal *> v = {&d, &c};
    for (const mammal *a : v) a->speak();
}

// Output:
// wan
// nyaa
```



Dynamic Polymorphism in C++

- Dynamic polymorphism is achieved by augmenting classes and structs (and sometimes unions).
- Use of **inheritance** to share interface/implementation between a parent class and child classes.
- Use of new keywords `virtual`, `override`, `final` to implement overriding member functions.
- `dynamic_cast<>` to safely cast types up and down the type hierarchy.
- First let's review some classic Object-Oriented Programming concepts.



OOP: Composition

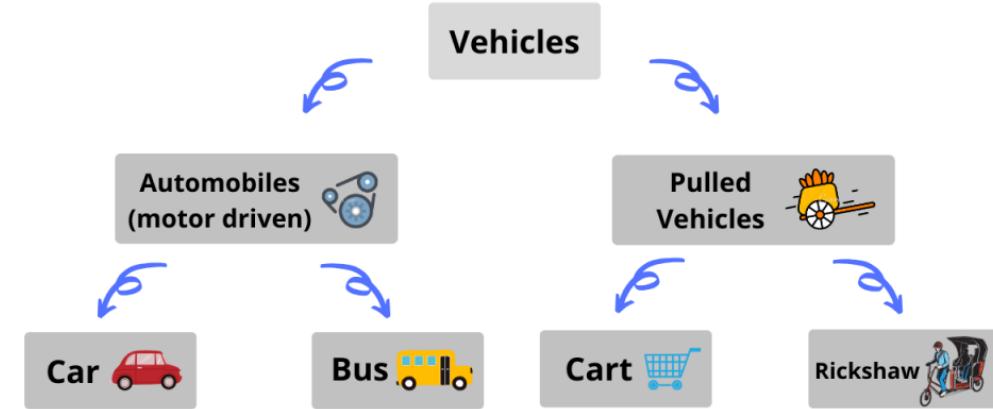
- Main idea of OOP: represent concepts as class-types.
- **Composition:** A contains a B but isn't a B itself.
 - "Has-a" relationship.
 - A person **has a** name;
 - A car **has a** battery;
 - Etc.
- Solves the problem of code duplication by keeping interfaces separate
 - Classes are coupled together by containment, however

```
class wheel {};  
  
class car {  
public:  
    /* Implementation */  
  
private:  
    wheel tl_;  
    wheel tr_;  
    wheel bl_;  
    wheel br_;  
};  
  
// A car “has” four wheels  
// delegate all operations that require  
// wheels to the wheel objects themselves.
```



OOP: Inheritance

- Main idea of OOP: represent concepts as class-types.
- **Inheritance:** A is a B and can do everything B does.
 - "is-a" relationship.
 - A dog **is an** animal.
 - A teacher **is an** employee.
 - Etc.
- Solve the problem of code duplication by sharing implementation and interface.
 - Can lead to incredibly difficult-to-manage type hierarchies, however.



Base class	Derived classes
Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Loan	CarLoan HomeImprovementLoan MortgageLoan
Employee	FacultyMember StaffMember
Account	CheckingAccount SavingsAccount

Inheritance in C++

- C++ supports multiple inheritance for all class-types.
 - Construction, destruction, and member look-up rules change when at least one base class exists.
- Inheritance kind depends on the **inheritance access specifier**.
- Implementation vs. Interface inheritance exists.
- Base class / derived class relationship expressed through inheritance.

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

struct hello {
    std::string msg1 = "hello!";
};

struct world {
    std::string msg2 = "world!";
};

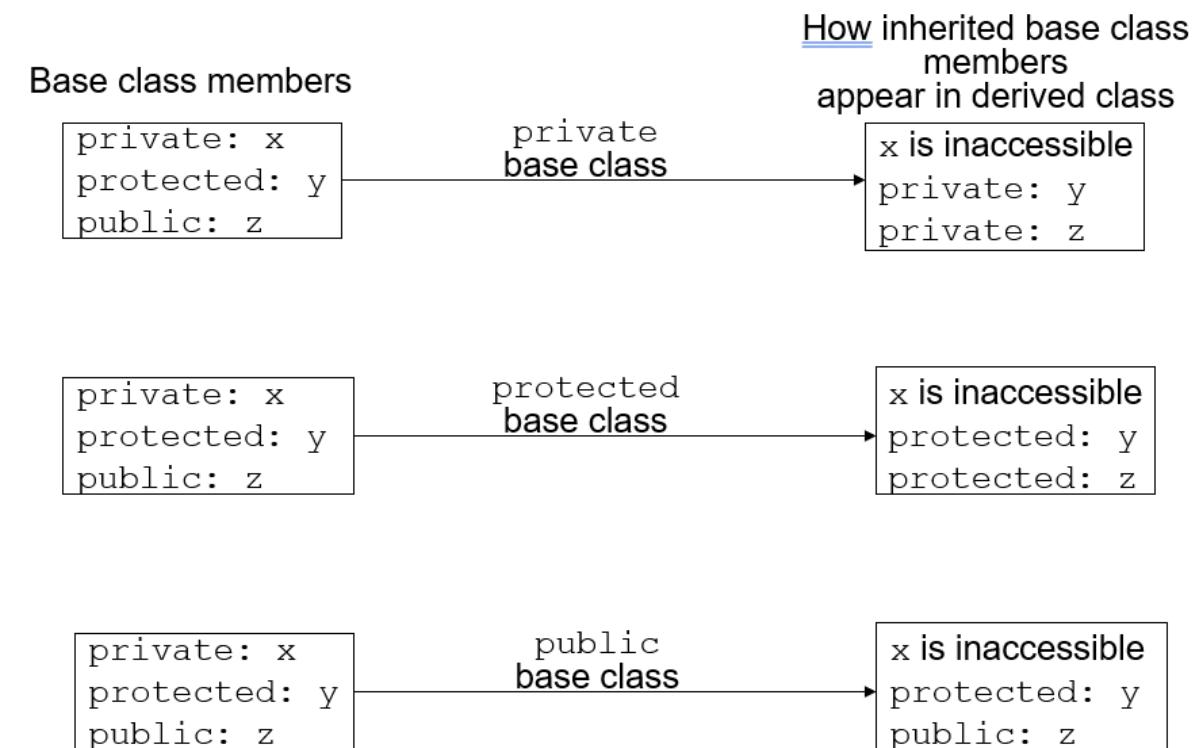
class child : public hello, public world {
public:
    auto greeting() const -> void {
        std::cout <<
        msg1 + " " + msg2 <<
        std::endl;
    }
};

int main() {
    // prints "hello world!"
    child{}.greeting();
}
```



Inheritance: Member Visibility

- Visibility is the maximum access exported from the deriving class.
- Visibility can be one of:
 - **public**
 - objects of the derived class can be treated as an object of the base class (generally use this unless you have good reason not to)
 - If you don't want public, you should (usually) use composition
 - **protected**
 - Derived class gains access to public and protected members of the parent
 - **private**
 - only accessible in the derived class.



Inheritance: Default Visibility

- The default inheritance visibility for classes is **private**.
- The default inheritance visibility for structs is **public**.

```
struct base {  
    int i;  
};  
  
class derived_class : base {  
    // base was derived from privately.  
    // this means that all non-private members  
    // in base are now “private” inside  
    // of derived_class  
};  
  
struct derived_struct : base {  
    // base was derived from publically.  
    // this means that all public members  
    // in base are still “public” in  
    // derived_struct, but “protected” members  
    // are still “protected” in derived_struct  
    // as well.  
};
```



Inheritance: Member Access

- It is possible after inheritance for a class-type to have multiple members with the same name.
- In that case, can use the scope operator (`:::`) to access the member of the specific class.
- If a member name is used *unqualified* and there is a collision:
 - For member functions with the same parameter list, will use the most derived class's overload if it exists.
 - Otherwise, if two base classes have the same method, then the call is ambiguous.
 - For data members, name collisions are always ambiguous.

```
#include <iostream>
struct uphill {
    int jack;
    int jill;

    void foo() {}
    void baz() {}
};

struct beanstalk {
    double jack;
    void baz();
};

struct two_worlds_collide : uphill, beanstalk {
    void foo() {
        // ERROR: jack is ambiguous
        std::cout << jack << std::endl;
    }

    void bar() {
        // OK: using uphill's jack
        std::cout << uphill::jack << std::endl;
        foo(); // calls two_worlds_collide::foo()
        baz(); // ERROR: call is ambiguous
    }
};
```



Inheritance: Interface vs. Implementation

- Interface inheritance is when only the interface of methods are intended to be inherited.
 - Does not mean implementation is not inherited also.
- Implementation inheritance is when the implementation are intended to be inherited.
- Inheritance kind depends on the **inheritance access specifier**.
 - Implementation inheritance uses *private* inheritance.
 - Anything other than private is interface inheritance.

```
#include <iostream>
struct base {
    auto greeting() const -> void {
        std::cout << msg << std::endl;
    }
    std::string msg = "hi!";
};

struct interface : public base {
    // no need to remake greeting()
};

struct implementation : private base {
    // need to re-create the greeting method
    auto greeting() const -> void {
        std::cout << msg + " world!\n";
    }
};

int main() {
    interface{}.greeting(); // prints "hi!"
    implementation{}.greeting(); // prints "hi, world!"
}
```

Inheritance: Construction

- Construction order changes when there is at least one base class.
- Each base class is constructed first (in order of inheritance), then this's data members, then the body of the constructor is run.
 - If a base class's constructor is not specified, then the default constructor is used.
 - A derived class cannot initialise fields in the base class.

```
#include <iostream>

struct A { A() { std::cout << "A " ; } };

struct B : A { B() { std::cout << "B " ; } };

struct C : A { C() { std::cout << "C " ; } };

struct alphabet : B, A, C {
    alphabet() : c{}, a{} {
        std::cout << "alphabetical";
    }

    C c;
    B b;
    A a;
};

int main() {
    alphabet alpha;
}

// output:
// A B A A C A C A B A alphabetical
```



Inheritance: Destruction

- Destruction order changes when there is at least one base class.
- First, `this`'s destructor runs.
- Then, the destructors of `this`'s data members run in the reverse order of declaration.
- Then, the destructors of `this`'s base classes run in the reverse order of inheritance.

```
#include <iostream>

struct A { ~A() { std::cout << "A\n"; } };

struct B : A { ~B() { std::cout << "B "; } };

struct C : A { ~C() { std::cout << "C "; } };

struct alphabet : A, B, C {
    ~alphabet() {
        std::cout << "alphabetical ";
    }

    B b;
    C c;
};

int main() {
    { alphabet alpha; }
}

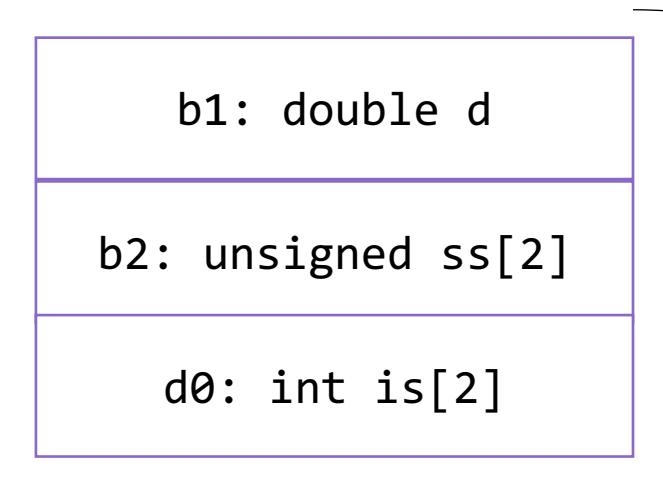
// output:
// alphabetical C A B A C A B A A
```

C++ Object Memory Layout

- Very important to understand how objects are laid out in memory.
- Base classes are laid out first (in order of inheritance).
- Then `this` is laid out.
- For deep type hierarchies, `this` may have data members which are very far apart from a base class's data members.

```
struct b1 { double d; };
struct b2: b1 { unsigned ss[2]; };
struct d0 : b2 { int is[2]; };

d0 d = {};
```



Object-Slicing Problem

- If a `b1` variable is declared on the stack *by value* how big should it be?
 - The compiler only knows its static type!
- When storing a class with base classes by value, classes lower in the hierarchy will not be stored since the compiler doesn't know about their existence.
- This is called the "object-slicing problem".

```
struct b1 { double d; };
struct b2: b1 { unsigned ss[2]; };
struct d0 : b2 { int is[2]; };

// SLICE! Compiler only copied the b1 part of d0{}
b1 b = d0{};
```

b1: double d

b2: unsigned ss[2]

d0: int is[2]

b's memory layout.

You can see how the non-b1 parts are sliced off when copying.



Polymorphic Classes

- **Polymorphism** means that a call to a member function will cause a different function to be executed depending on the runtime type of the object.
- Polymorphism allows reuse of code by allowing objects of related types to be treated the same.
- Polymorphism in C++:
 - Static (compile-time) type vs. dynamic (runtime) type.
 - Overridable methods through the `virtual`, `override`, and `final` keywords.
 - Due to the Object Slicing Problem only **pointers** and **references** to classes can exhibit polymorphic behaviour.



Static vs. Runtime Type of Polymorphic Classes

- Static type is the type it is declared as in the source code.
- Dynamic type is the type of the object at run-time.
- Due to object slicing, value objects **always** have the same static and dynamic type.

```
struct base {};
struct derived : base {};

int main() {
    auto b = base{};
    auto d = derived{};

    base sliced = d; // not good, don't do this!

    // The following could all be replaced with
    // pointers and have the same effect.
    const base &base2base = b;

    // A potential reason to use auto:
    // you can't accidentally do this.
    const base &base2derived = d;

    // Fails to compile
    const derived &derived2base = b;

    const derived &derived2derived = d; // OK!

    // Fails to compile despite a ref to a derived class
    const derived &derived2base2derived = base2derived;
}
```



Static vs. Runtime Binding

- Static binding: Decide which function to call at compile time.
 - Based on static type in the source code.
- Dynamic binding: Decide which function to call at runtime.
 - Based on dynamic type.
- C++ is by default statically typed.
 - Types are specified at compile time.
 - Static binding for non-virtual functions.
 - Dynamic binding for virtual functions.
- Very different from almost all other languages that support OOP!



virtual Methods

- How does the compiler know which method to call in a polymorphic class?
- Explicitly tell compiler that a method is meant to be overridden in a subclass.
- Use the `virtual` keyword in the base class:
 - For methods in the base class that expect to be overridden in derived class.
 - Dynamic binding: actual function to call bound at runtime by the compiler.
 - It ensures that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
 - Static binding: Without virtual member functions, actual function to call determined at compile-time.
- Use the `override` keyword in the derived class

```
#include <iostream>

struct cat {
    Virtual void speak() const {
        std::cout << "meow\n";
    }
};

struct garfield : cat {
    void speak() const override {
        std::cout << "I want lasagne!\n";
    }
};

int main() {
    garfield g;
    cat c;

    const cat &cg = g;
    const cat &cc = c;

    cg.speak(); // prints "I want lasagne!"
    cc.speak(); // prints "meow"
}

// though cg and cc have the same static type,
// at runtime their types are different.
// The correct function to call is looked up
// at runtime
```

The override Keyword

- Tells the compiler this method overrides a virtual function in a base class.
- While `override` isn't required by the compiler, you should **always** use it.
- `override` fails to compile if the function doesn't exist in the base class. This helps with catching errors related to:
 - Refactoring / typos.
 - `const` / non-`const` methods.
 - Slightly different signatures.

```
struct character {  
    virtual int power() const { return 6771; }  
};  
  
struct guardian : character {  
    // ERROR: this method does not override a  
    // virtual method in `character`  
    // (missing const qualification)  
    int power() override { return 42; }  
};  
  
struct vegeta : character {  
    // OK: matches int power() const in `character`  
    int power() const override { return 9001; }  
};
```

The final Keyword

- Specifies to the compiler that a method cannot be overridden in a derived class.
 - This means static binding if you have a reference/pointer to a derived class, but dynamic binding for a base class reference/pointer.
- Also specifies to the compiler that a class-type cannot be derived from.

```
struct top {
    virtual void greet() const final {
        std::cout << "hello, world!" << std::endl;
    }
};

struct final middle : top {
    // ERROR: cannot override `greet`:
    // it has declared as final
    void greet() const override {
        std::cout << "heyyy" << std::endl;
    }
};

// ERROR: cannot derive from `middle`:
// it has been declared as "final"
struct bottom : middle {};
```



virtual Methods & Default Arguments

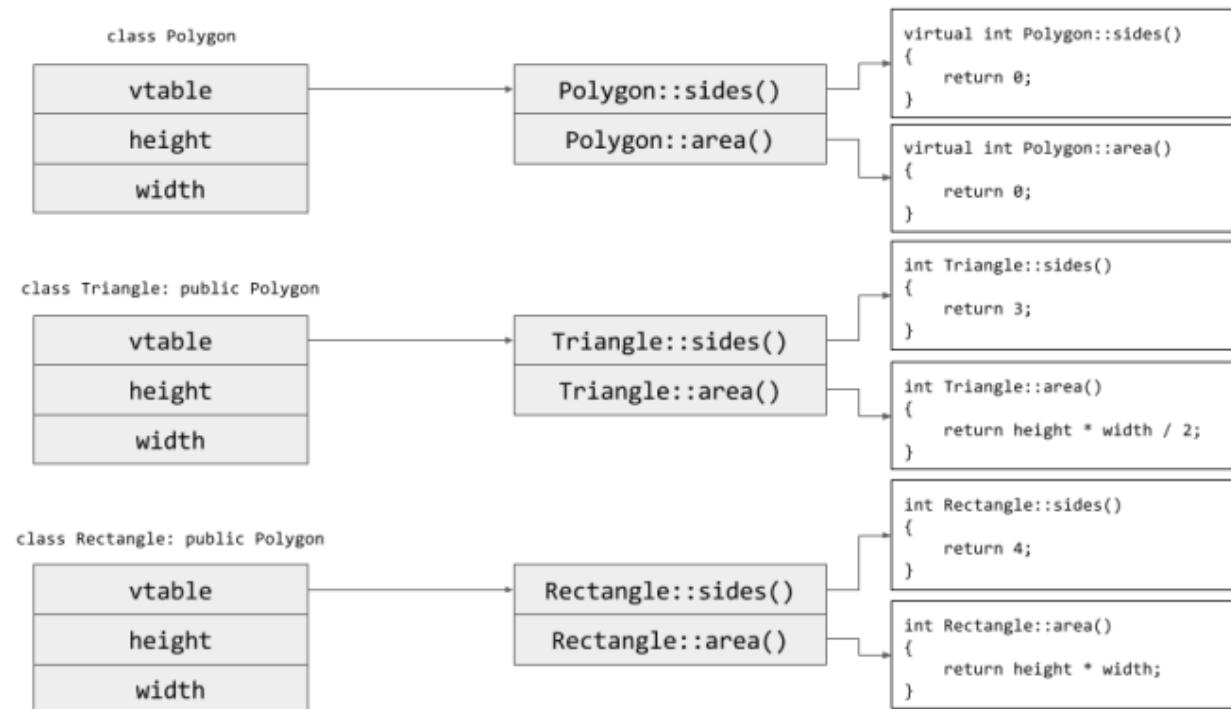
- Default arguments are determined at compile-time for efficiency reasons.
- Hence, default arguments need to use the **static** type of the function.
- **Avoid default arguments when overriding virtual functions.**

```
struct base {  
    virtual void print_num(int i = 1) {  
        std::cout << "Base " << i << '\n';  
    }  
};  
  
struct derived: base {  
    void print_num(int i = 2) override {  
        std::cout << "Derived " << i << '\n';  
    }  
};  
  
int main() {  
    derived d;  
    base *b = &d;  
  
    // Prints "Derived 2"  
    d.print_num();  
  
    // Prints "Derived 1" because the default argument  
    // was chosen due to b's static type (base)  
    // though the actual function to call was chosen due  
    // to b's dynamic type (derived)  
    b->print_num();  
}
```



How virtual works (VTables)

- Each polymorphic class has a vtable stored in the text segment of the binary.
 - A vtable is an array of function pointers.
 - Compiler hashes the names of virtual methods and stores a pointer to this class's *specific* implementation of that virtual or copies the parent's corresponding function pointer if not overridden.
- If the vtable for a class is non-empty, then every member of that class has an implicit data member that is a pointer to the vtable.
- When a virtual function is called **on a reference or pointer type**, then the program does the following:
 - Follow the vtable pointer to get to the vtable .
 - Increment by an offset (calculated by the compiler), which is a constant.
 - Call the function pointer pointed to by vtable[offset].



[Another example here](#)



Constructing Polymorphic Objects

- Virtual methods cannot be used until a class is fully constructed.
- A base class's virtual methods can be used once its constructor has run.
- Due to objects being stored inline, if you want to store a polymorphic object, use a pointer.
 - Storing references in classes immediately makes the class non-copyable (since references cannot be rebound).
 - If you want to store a reference, use `std::reference_wrapper`

```
// would work in a language like Java
// will NOT work in C++
auto base = std::vector<BaseClass>{};
base.push_back(base{});
base.push_back(derived1{});
base.push_back(derived2{});
```

```
// Idiomatic C++ code
auto base = std::vector<std::unique_ptr<base>>{};
base.push_back(std::make_unique<base>());
base.push_back(std::make_unique<derived1>());
base.push_back(std::make_unique<derived2>());
```

Destructuring Polymorphic Objects

- Virtual methods cannot be used if a class is partially destructed.
- Every polymorphic class **must** have a virtual destructor so the resources are destructed in a proper order when you delete a base class pointer pointing to derived class object.
 - If the base class destructor is **virtual**, derived class's destructors are automatically **virtual**.
 - Remember: When you declare a destructor, the move constructor and assignment are not synthesised.
 - **Forgetting this can be a hard bug to spot.**

```
#include <iostream>
#include <memory>

struct base {
    base() { std::cout << "A ";}
    base(base &&) = default;
    base &operator=(base &&) = default;
    virtual ~base() { std::cout << "B\n"; }
};

struct derived: base {
    derived() { std::cout << "C ";}
    derived(derived &&) = default;
    derived &operator=(derived &&) = default;
    ~derived() override { std::cout << "D ";}
};

int main() {
    std::unique_ptr<base>{new derived{}};
}

// Output: A C D B
```

Pure Virtual Methods

- Virtual functions are good for when you have a default implementation that can be overridden.
- Sometimes there is no good default behaviour.
- A **pure virtual function** specifies a function that a class **must** override.
- Potentially the most arcane syntax in all of C++.
- Non-assessable extra reading: [\(Im\)pure virtual functions](#)

```
struct canvas { /* implementation */ };

struct shape {
    // Derived classes may forget to override this.
    virtual void draw(canvas &) {}

    // Fails at link time because
    // there's no definition.
    virtual void draw(canvas &);

    // Pure virtual function.
    // Any derived class must override this.
    // Declare a virtual method as normal
    // and "set" it to 0.
    virtual void draw(canvas &) = 0;
};

struct circle : shape {
    void draw(canvas &c) override { /*...*/ }
};
```

Rules for virtual Methods

1. Virtual member functions cannot be static.
2. Virtual member functions cannot be friends.
3. Virtual member functions only exhibit dynamic binding when used through a pointer or reference.
 - This includes pointers/reference to the most-derived type, or to any base class in the type hierarchy.
4. The prototype of virtual functions must be the same in the base as well as derived class.
 - Ensure this with `override`.
5. They are always declared in the base class and overridden in the derived class. It is not mandatory for the derived class to override virtual methods (unless pure virtual).
 - In that case, the base class version of the method is used.
6. A class must have a virtual destructor but it cannot have a virtual constructor.



Types of Class Methods

Syntax	Name	Meaning
<code>virtual void fn() = 0;</code>	Pure virtual	Inherit interface only
<code>virtual void fn() {}</code>	Virtual	Inherit interface with optional implementation
<code>void fn() {}</code>	Non-virtual	Inherit interface and mandatory implementation

Note: non-virtuals can be hidden by writing a function with the same name in a subclass.
DO NOT DO THIS.



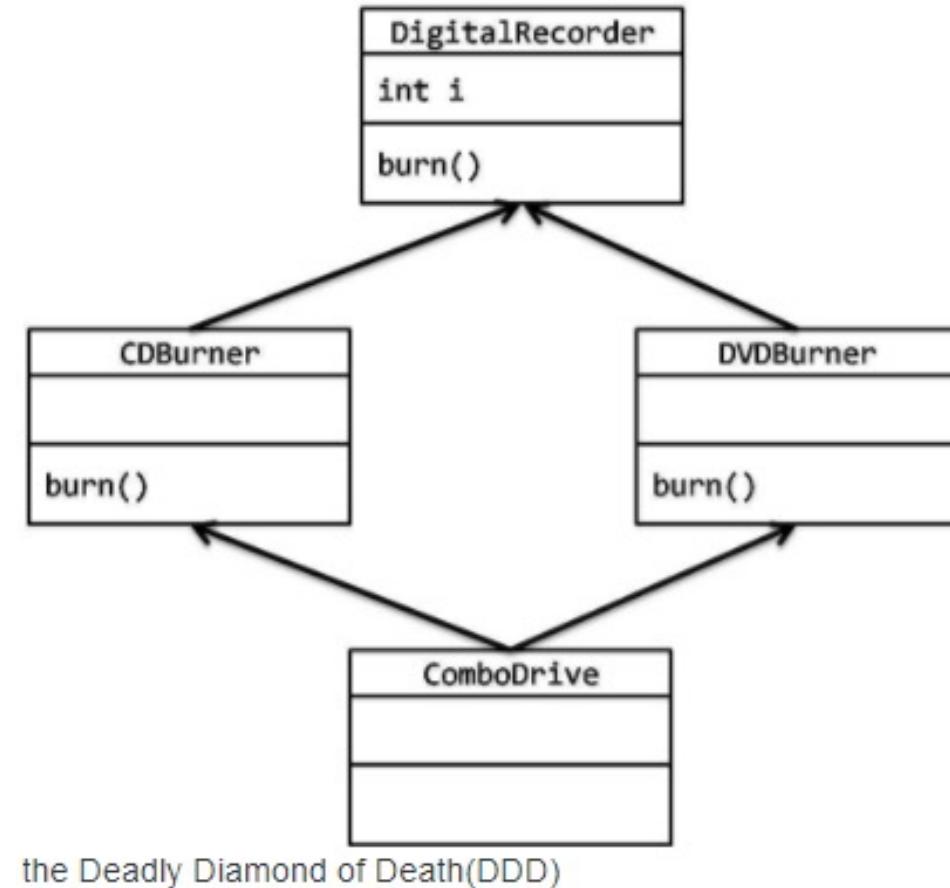
Abstract Base Classes (ABC)

- Might want to deal with a base class, but the base class by itself does not make sense.
 - E.g.: What is the default way to draw a shape? How many sides by default?
- Might want some default behaviour and data, but derived classes should fill out the rest of the behaviour.
 - E.g., all files have a name, but are reads done over the network or from a disk.
- If a class has at least one pure virtual method, the class is abstract and cannot be constructed.
 - It can, however, have constructors and destructors.
 - These provide semantics for constructing and destructing the ABC subobject of any derived classes.



The Deadly Diamond (of Death)

- C++ supports multiple inheritance.
- This can create a few specific problems:
 - In deep hierarchies, it is possible for a derived class's ancestors to inherit from the same class!
 - The derived class could have two or more copies of the same base class.
 - Calls to unqualified member functions are ambiguous.
- This is known as the "deadly diamond".
 - Not specific to C++.
 - Any OOP language that supports interfaces (Java, Python) also has this problem.
- C++ solves this via virtual inheritance.



virtual Inheritance

- If a derived class inherits virtually from a base class, it is guaranteed to only have **one** copy of any shared ancestors.
- All ancestors in a hierarchy *must also* inherit that base virtually, however.
- All virtual base class subobjects are initialised first before non-virtual ones.
- Unqualified calls to member functions with name collisions go through overload resolution as if those member functions were declared **virtual**.
- Without virtual inheritance, the call would be ambiguous.

```
struct B { int n; };
class X : public virtual B {};
class Y : virtual public B {};
class Z : public B {};
// every A has one X, one Y, one Z, and two B's:
// - one that is the base of Z
// - and one that is shared by X and Y
struct A : X, Y, Z {
    A() {
        // modifies the virtual B subobject's member
        X::n = 1;
        // modifies the same virtual B subobject's member
        Y::n = 2;
        // modifies the non-virtual B subobject's member
        Z::n = 3;
        // prints 223
        std::cout << X::n << Y::n << Z::n << '\n';
    }
};

struct M { void f(); };
struct B1: virtual M { void f(); };
struct B2: virtual M {};

struct C : B1, B2 {
    void foo() {
        X::f(); // OK, calls X::f (qualified lookup)
        f(); // OK, calls B1::f (unqualified lookup as if virtual)
    }
};
```



Casting Up a Type Hierarchy

- Casting from a derived class to a base class is called up-casting.
- This cast is always safe.
 - All derived classes are base classes, after all.
- Because the cast is always safe, C++ allows this as an implicit cast.
- One potential reason to use `auto` is that it avoids implicit casts.

```
struct animal { /* ... */ };
struct dog : animal { /* ... */ };

int main() {
    auto doggo = dog();
    // Up-cast with references.
    animal& animalia_r = doggo;
    // Up-cast with pointers.
    animal* animalia_p = &doggo;
}
```

Casting Down a Type Hierarchy

- Casting from a base class to a derived class is called down-casting.
- This cast is not guaranteed to be safe.
- The compiler doesn't know if an animal happens to be a dog.
 - If you **know** it is, you can use `static_cast`.
 - Otherwise, you can use `dynamic_cast`:
 - Returns null pointer for pointer types if it doesn't match.
 - Throws exceptions for reference types if it doesn't match.
- `dynamic_cast` relies on **Runtime Type Information (RTTI)**.
 - RTTI is one of the most disabled features of C++ due to its performance cost.

```
struct animal { virtual ~animal() = default; };
struct dog : animal { /* ... */ };
struct cat : animal { /* ... */ };

dog d;
cat c;
animal& ad = d;
animal& ac = c;

int main() {
    // Attempt to down-cast with references.
    dog& dr1 = static_cast<dog&>(ad);
    dog& dr2 = dynamic_cast<dog&>(ad);

    // Undefined behaviour - incorrect static cast.
    dog& dr3 = static_cast<dog&>(ac);

    // Throws exception
    dog& dr4 = dynamic_cast<dog&>(ac);

    // returns null pointer
    dog* dp1 = dynamic_cast<dog*>(&ac);
}
```

OOP: Covariance

- If a derived class overrides a virtual method from a base class, what should the return type be?
- Every possible return type for the derived's overridden method must be a valid return type for the base's original method.

```
struct fruit { virtual ~fruit() = default; };
struct apple : fruit { /* ... */ };
struct granny_smith : apple { /* ... */ };

struct parent {
    apple app = apple{};

    virtual const fruit &get_fruit() const {
        // OK: apple is a fruit!
        return app;
    }
};

struct child : parent {
    granny_smith gs = granny_smith{};

    const apple &get_fruit() const override {
        // OK: granny_smith apples are a fruit too!
        // this method override is covariant
        return gs;
    }
};
```



OOP: Contravariance

- If a derived class overrides a virtual method from a base class, what should the parameter types be?
- An overridden method can accept more general types as the parameters.
- Every possible argument to the base class's method **must** be a valid argument to the derived's overridden method.
- Not as easy to ensure as covariant methods in C++.

```
struct fruit { virtual ~fruit() = default; };
struct apple : fruit { /* ... */ };

struct parent {
    virtual void eat(const apple &a) const {
        // nom nom nom on that apple
    }
};

struct child : parent {
    // this method override is contravariant!
    // ...but the compiler doesn't accept this
    // because the signatures between the
    // base and derived class don't match.

    // could potentially reuse the base's
    // method if we explicitly down-cast
    // with dynamic_cast or static_cast
    void eat(const fruit &f) const override {
        // nom nom nom on that fruit
    }
};
```



Can Inheritance be dangerous?



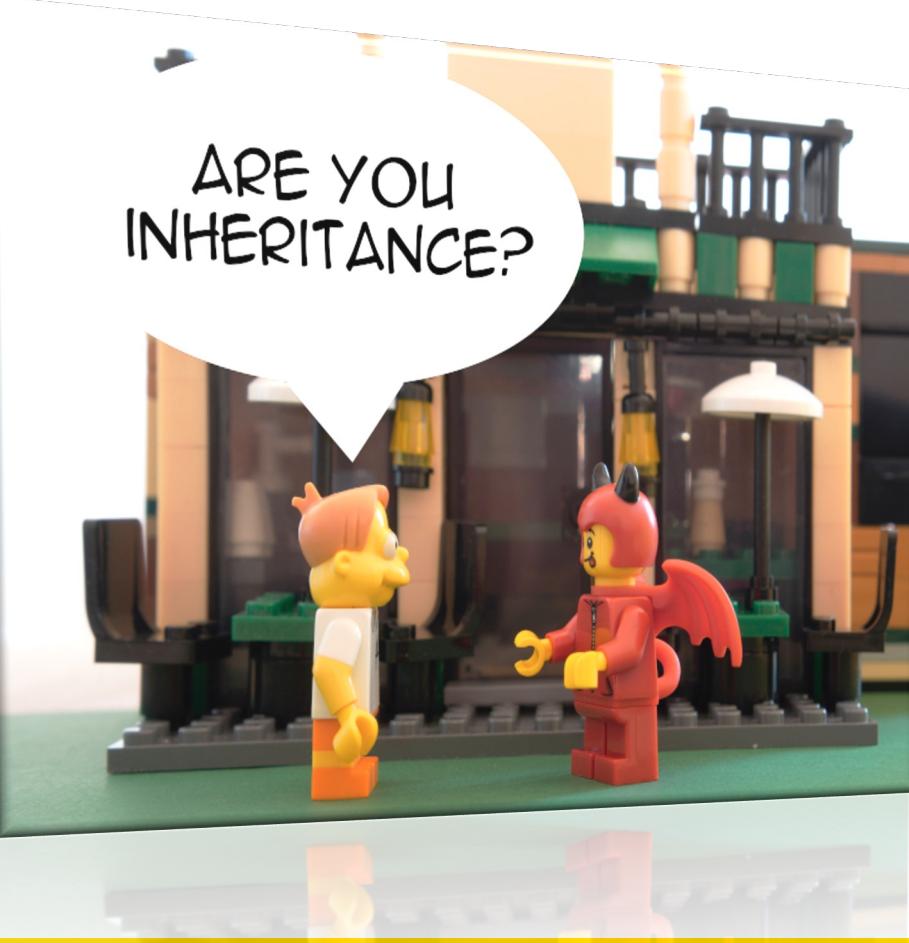
UNSW
SYDNEY

Inheritance-the base class of evil

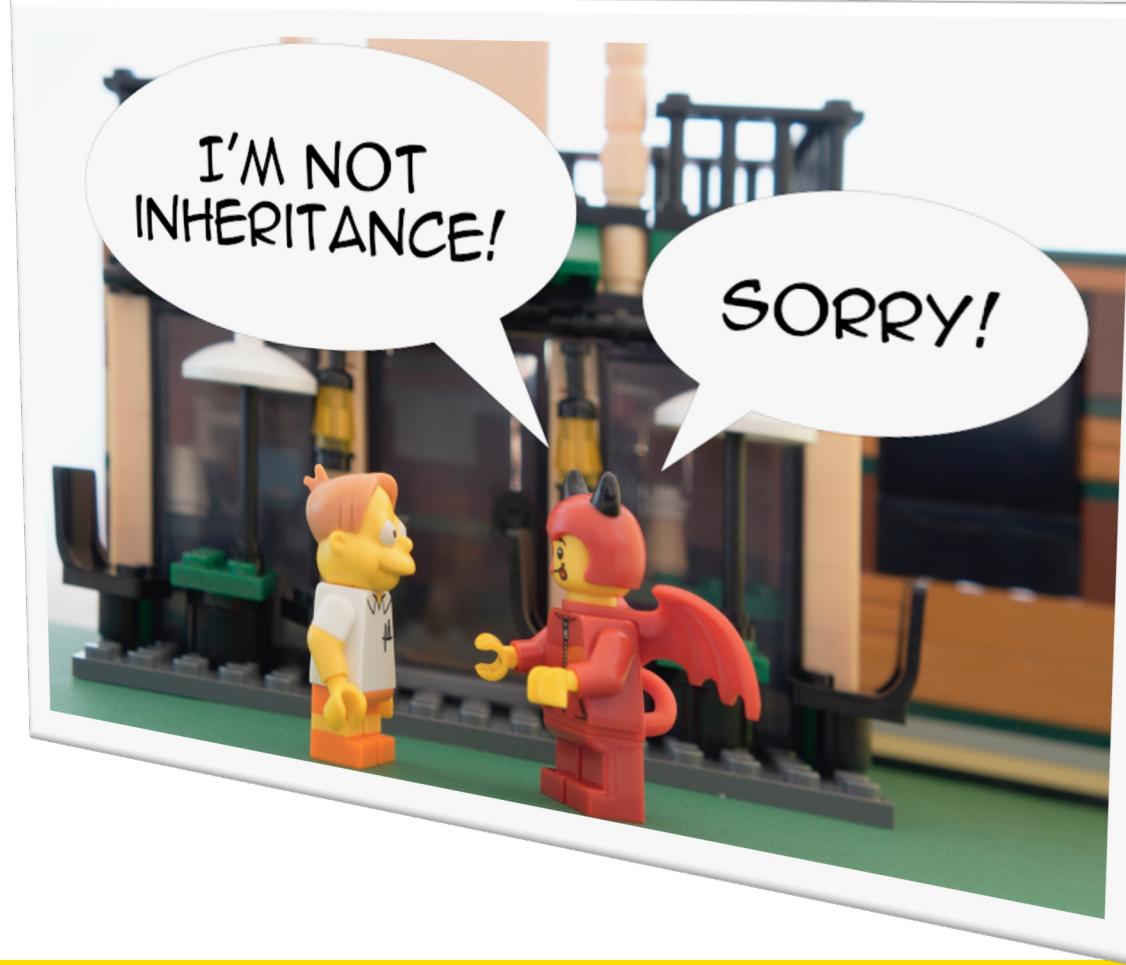
Sean Parent)

<https://learn.microsoft.com/en-us/events-goingnative-2013/inheritance-base-class-of-evil>

<https://thevaluable.dev/guide-inheritance-oop/>



ARE YOU
INHERITANCE?



I'M NOT
INHERITANCE!

SORRY!

Problem with Inheritance

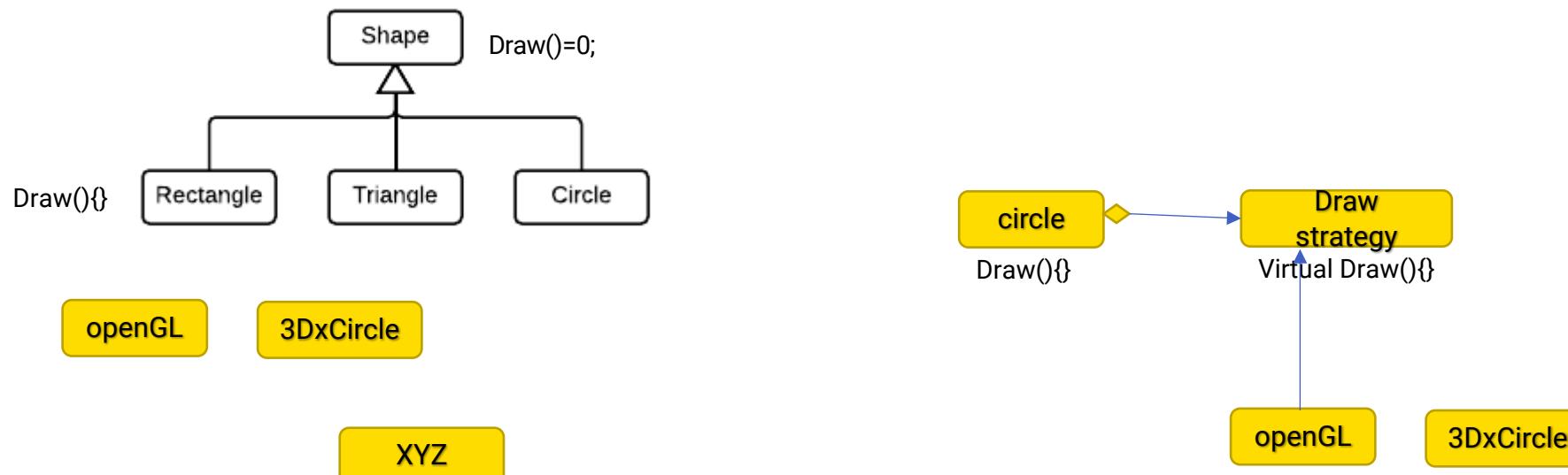
- Often requires dynamic allocation.
- Ownership and nullability semantics become hazy.
- Intrusiveness: requires modifying child classes.
- No more value semantics.
- Can change semantics for algorithms and containers.



Dependencies: Design Principle

Problem-Change: Software must be adaptable

- Using inheritance is not the only way to extend a class behavior, but definitely is the **most dangerous and harmful one**.
- One change in the base class could affect the behavior of the child.



Some Guidelines

- Use **only** two type of class-level declarations: **interfaces and final classes**;
- Inject interfaces in dependent classes' constructors;
- **Don't allow** any class dependency to be injected other than interfaces;
- Use a Dependency Injection Container (or an equivalent method, depending on which language you're coding with) to handle the creation of my instances;
 - If injecting too many dependencies in a class, rethink design in terms of class responsibilities and using the interface segregation principle;
- When required, split complex behavior in multiple final classes implementing the same interface;
- Use inheritance **only** when it makes sense on a semantic level and only for extension purposes, without any base behavior change;



Feedback (stop recording)



UNSW
SYDNEY

COMP6771

Advanced C++ Programming

8.1 Static Polymorphism

In This Lecture

Why?

- C uses #define or void* (a.k.a, is weakly-typed).
- C++ is strongly-typed.
- Understanding compile time polymorphism in the form of templates helps understand the workings of C++ on generic types.

What?

- All the different templates in C++
- Template parameters (type, non-type, template template)
- Mechanics of Templates

Recommended Reference:

- C++ Templates: The Complete guide (David Vandevoorde, et al. 2018)

Motivation for Templates

- Question: how to remove code duplication for functions with the same logic but that operate on different types?
 - This branch of programming is called *Generic Programming*.
- **Generic Programming:** Generalising software components to be independent of a particular type
 - STL is a great example of generic programming
- Without generic programming, to create two logically identical functions that behave independently of types, we have to rely on function overloading.
- Templates is how Generic Programming is implemented in C++ to be *fast* and *flexible*.

```
// in C.  
// only works because of literal cut & paste!  
// there is no type checking here at all!  
#define min(x, y) ((x) < (y) : (x) : (y))  
  
// in C++ before templates  
int &min(int &x, int &y) {  
    return x < y ? x : y;  
}  
  
const int &min(const int &x, const int &y) {  
    return x < y ? x : y;  
}  
// repeat for the other types you care about...  
  
// in C++ WITH templates  
// one function template to rule them all!  
T &&min(T &&x, T &&y) {  
    return x < y ? x : y;  
}
```

Templates Overview

- C++ has **five** kinds of templates:
 - Function templates
 - Class templates
 - Alias templates (since C++11)
 - Variable templates (since C++14)
 - Variadic Templates (since C++11)
- Each template is a recipe for the compiler to generate a usable entity of that kind.
 - E.g., a function template can be used to *synthesise* a function.
 - It itself is **not** a function!
- Templates can also be used for metaprogramming (next week).

Template Parameters

Three kinds of template parameters.

- **Type** template parameters:
 - Each template parameter holds a type name.
- **Non-type** template parameters:
 - Each template parameter holds a compile-time knowable value.
- **Template template** parameters:
 - Each template parameter holds the name of a template.
- All template parameters can be given default values.
- Template parameters can also be anonymous (no name).

```
template < // template parameter list
          // a type parameter
          // defaults to int
          typename T = int
>
auto min(T &&x, T &&y) -> T&& {
    return x < y ? x : y;
}

// each occurrence of `T` above is
// replaced by the concrete type
// at compile time.
```

Type Template Parameters

- A **type variable** that holds a type.
- Created by either using `typename` or `class` in the template parameter list.
 - Before C++11, only `class` could be used.
 - Now, prefer using `typename`.
 - No difference between the two, just a name change.
- Name style: PascalCase.

```
// below is the template signature for
// std::vector

namespace std {
    template <
        // element type, T
        // using `typename` `typename` T,
        // allocator, for dynamic memory
        // allocation
        // using pre-C++11 `class` syntax
        class Allocator
    >
    class vector { /* ... */ };
}
```

Non-type Template Parameters

- An implicit `constexpr` **variable** that holds a value at compile-time.
- Restrictions on what can be a non-type template parameter (as of C++20):
 - Integral types (`int`, `char`, `long`, etc.)
 - Floating point types (`float`, `double`)
 - Pointers and references (including function pointers)
 - Literal structural types
 - Essentially, C-style structs optionally with base classes
- As of C++20, it is possible to let the compiler deduce the type of the non-type template parameter with `auto`.

```
template <int I>
int sub(int j) { return I - j; }

template <float F>
float identity() { return F; }

template <void(*Fn)(int)>
auto invoke(int i) { return Fn(i); }

struct base { int i; };
struct derived : base { int j; };

template <derived D>
void sum() { std::cout << D.i + D.j << std::endl; }

auto course = sub<6771>(0);
auto pi = identity<3.14f>();
auto result = invoke<sub<6771>>(42);
constexpr derived d = {};
auto summed = sum<d>();
```

Template Template Parameters

- A template parameter that holds the name of a **template**.
- The number of template parameters the template itself takes must be known.
 - The number must be exact; no default arguments can be used.
 - It is not necessary to name the template template parameter's template parameters.

```
// the definition of std::stack
// probably looks something like this

namespace std {
    template <
        // element type of the stack
        typename T,
        // the underlying sequential container
        // no need to give a name to Container's
        // template parameter - it cannot be used.
        template <typename>
        typename Container
    >
    class stack {
        // other implementation...
        Container<T> cont_;
    };
}
```

Default Template Parameters

- We can provide default arguments to template parameters.
- The set of default template arguments accumulates over all declarations of a given template.
- Once a default is given, all subsequent parameters must also have default values.
 - Matches the same rules as default function arguments.

```
template <typename>
struct empty {};
```



```
template <
    typename T = int,
    auto I = 0,
    template <typename>
    typename Container = empty<T>
>
struct eclectic {};
```

Function Templates

- Function template: not actually a function.
 - Generalisation of algorithms.
- A blueprint for the compiler to synthesise particular instances of a function varying by type.
 - Single definition that can generate many definitions.
 - The compiler *instantiates* a function template only when a function by that name is needed and no proper function exists.

```
#include <iostream>

template <typename T>
auto add_or_concat(const T &a, const T &b) -> T {
    return a + b;
}

int main() {
    // prints 3
    std::cout << add_or_concat(1, 2) << std::endl;

    // prints "hello world"
    std::string h = "hello", w = "world";
    std::cout << add_or_concat(h, w) << std::endl;

    // code is only generated for int and std::string
    // no other type was used, so no other version
    // was instantiated.

    return 0;
}
```

Function Template Argument Deduction

- To instantiate a function template, all the template parameters must be known.
 - But they don't need to be specified!
- The compiler will attempt to deduce any missing parameters from the function arguments.
- **Important:** implicit conversions:
 - For *type parameters*, implicit conversions do **not** occur.
 - For non-type template parameters, the compiler will perform implicit conversions.
- Some notable rules:
 - Most specific type is matched.
 - May match on type specifiers (const, volatile)
 - May match on modifiers (pointer *, reference &)
 - [Complete list of rules here](#)

```
template <typename T>
auto min(T &&x, T &&y) { return x < y ? x : y; }

template <typename T, std::size_t N>
T sum_array(const T(&arr)[N]) {
    return std::accumulate(arr, arr + N, T{});
}

int main() {
    // OK: T = int
    min(1, 2);
    // NOT OK: T = const char *; compare pointers
    min("hello", "world");

    // OK: T = std::string
    min(std::string{"hello"}, std::string{"world"});
}

int nums[] = {3, 2, 1};
// OK: T = int, N = 3
std::cout << sum_array(nums) << std::endl;
}
```

auto Revisted

- Ever wondered what rules auto uses to deduce types?
 - auto uses the same rules as function template argument deduction!
- As of C++14, can use auto as lambda function parameters.
 - This creates a *generic* lambda.
 - A new lambda is created for each argument type combination.
- As of C++20, auto can also be used in function parameters.
 - This creates an *implicit* function template.
 - A new function is created for each argument type combination.

```
// generic lambda
auto min = [](<const auto &x, const auto &y) {
    return x < y ? x : y;
}

// C++20-style function template
auto max(const auto &x, const auto &y) {
    return y < x ? y : x;
}

int main() {
    // prints 1, auto deduced to be int
    std::cout << min(1, 2) << std::endl;

    // prints 'a', auto deduced to be char
    std::cout << max('0', 'a') << std::endl;

    // won't compile: auto deduced const char *, int
    // pointers are not comparable to integers
    std::cout << max("hi", 6771) << std::endl;

    // prints 'a', auto deduced char, int
    // char implicitly convertible to int
    std::cout << min('a', 127) << std::endl;
} // altogether, 2 versions of min, 1 version of max
```

Explicit Template Argument Deduction

- If we need more control over the normal deduction process, we can explicitly specify the types being passed in.
- This will allow for implicit conversions of the passed arguments to the explicitly-stated types.
- It is possible to explicitly state a subset of the template parameters.
 - The remaining template parameters undergo normal template argument deduction.

```
template<typename T, typename U>
auto min(T a, U b) {
    return a < b ? a : b;
}

auto main() -> int {
    auto i = 0;
    auto d = 3.0;
    // int min(int, double)
    min<int>(i, d);
    // int min(int, int)
    min<int, int>(i, static_cast<int>(d));

    // double min(double, double);
    min<double>(static_cast<double>(i), d);

    // double min(double, double)
    min<double, double>(i, d);
}
```

Overload Resolution Revisited

- The compiler changes how it performs overload resolution when function templates are involved:
 1. The compiler constructs the overload candidate set first *from real functions*.
 - This includes previously instantiated function templates.
 2. If there is no best match, then it will instantiate a new function with the appropriate types from the template.
 - It is important to remember function templates are *not* part of the overload set. Only functions synthesised from the template are.

```
template <typename T>
void printer(const T *ptr) {
    std::cout << ptr << std::endl;
}

void printer(const int *ptr) {
    std::cout << ptr << std::endl;
}

int main() {
    int i = 0;
    double d = 0.0;

    // no function void printer(const double *)
    // synthesise one
    printer(&d);

    // found void printer(const int *)
    // don't even consider the template
    printer(&i);

    // found previous instantiation
    // void printer(const double *). Use that
    printer(&d);
}
```

Class Templates

- Similar to function template, a class template is a blueprint for synthesising a class-type.
- Is not actually a class.
- All of the members inside of the class template are parameterised based on the template parameters.

```
// Before templates...
struct vec3i {
    int(&)[3] get_elems() const { /* ... */ }
    int elems[3];
};

struct vec4d {
    double(&)[4] get_elems() const { /* ... */ }
    double elems[4];
};

// ...after templates!
template <typename T, std::size_t N = 3>
struct vec {
    T(&)[N] get_elems() const { /* ... */ }
    T elems[N];
};
```

Member Function Templates

- Usually the member functions of a class template are parameterised on the class template's template arguments.
- It is possible to make the member function itself a template.
 - In this case, it would have two sets of parameters: the class template's and its own.
- This is useful for creating many overloads of a member function.
 - E.g., converting constructors.

```
// std::vector's iterator constructor probably
// looks something like this

namespace std {
    template <typename T, /* others... */>
    class vector {
        template <typename InputIt>
        vector(InputIt first, InputIt last) {
            // allocate memory for an array
            // copy elements between first and last
            // into the allocated array
        }
    };
}

// another great example is std::set's
// transparent comparator feature for .find().
// read more about it here
// (notably overloads 3 & 4)
}
```

Static Members of a Class Template

- Possible to create static data members for a class template.
 - Every instantiation has its own version of the static member.
 - Initialisation also looks like a template.
 - Can be defined inline if constant.
- Also possible to create static member functions.

```
template <typename T>
struct t_is_small {
    // inline constant static data member
    static constexpr bool value = sizeof(T) <= 4;
};

template <typename T>
class rational {
public:
    // static data member
    static std::optional<rational> null;

    // static member function
    static auto make_rational(T n, T d) {
        /* implementation... */
    }

    // other implementation details...
};

// out-of-line defintion of static data member.
template <typename T>
std::optional<rational> rational<T>::null = {};
```

Friends of a Class Template (error)

- Class templates can also have friends.
- **Caution:** the friend declaration declares a non-template function (see example on the right).
 - To make a friend based on the class template's parameters, the friend itself also needs to be a template.
 - Best to make all friends hidden friends to avoid confusion.

```
// This looks like it would work, and it will even compile.  
// However, the linker will fail to find operator<<.  
template <typename T, int N>  
class vec {  
public:  
    friend std::ostream &operator<<(  
        std::ostream &os, const vec &v  
    );  
  
private:  
    T elems_[N];  
};  
  
template<typename T, int N>  
std::ostream &operator<<(  
    std::ostream &os, const vec<T, N> &v  
) {  
    return os;  
}  
  
int main() {  
    vec<int, 3> v;  
    std::cout << v << std::endl;  
}
```

Friends of a Class Template (fixed)

- Class templates can also have friends.
- **Caution:** the friend declaration declares a non-template function (see example on the right).
 - To make a friend based on the class template's parameters, the friend itself also needs to be a template.
 - Best to make all friends hidden friends to avoid confusion.

```
// To fix, the class template and the friend template are
// declared first...
// Then a <> is added after the function name in the friend
// Finally, the friend is defined below.
// Avoid this - use hidden friends.

std::ostream &operator<<(std::ostream &os, const vec<T, N> &v);

template <typename T, int N>
class vec {
public: friend std::ostream &operator<< <>(
    std::ostream &os, const vec &v
);

private:
    T elems_[N];
};

template<typename T, int N>
std::ostream &operator<<(std::ostream &os, const vec<T, N> &v) {
    return os;
}
```

Class Template Argument Deduction

- NEW in C++17: Class Template Argument Deduction
 - Also called CTAD for short.
- Equivalent to function template argument deduction, but for class templates.
 - Same rules about not doing implicit conversions also apply.
- User must define the rules for deducing template arguments from a constructor call.
 - Then, the user doesn't need to specify the class template's parameters when declaring variables.
 - Syntax looks like the constructor signature.

```
template <typename T>
class rational {
public:
    rational(T num, T denom);

private:
    T num_;
    T denom_;
};

// CTAD definition.
// When the rational(T, T) constructor would have been
// used, then deduce the template parameter to be T.
template <typename T>
rational(T, T) -> rational<T>;

int main() {
    // from initialiser, rat inferred to be rational<int>
    rational rat = {1, 2};

    // error: deduced two conflicting types!
    rational err = {1, 2.3};
}
```

Out-of-line Definitions

- A class template's methods can be defined out-of-line.
- For member templates, there are two sets of template parameters.
- Unless you have good reason to, prefer defining methods inline in the template definition.

```
template <typename T>
struct bar { /* definition... */ };

template <typename T>
struct foo {
    template <typename U>
    foo(const bar<U> &b);

    void baz();
};

template <typename T> // top-level template first
template <typename U> // bottom-level template 2nd
foo<T>::foo(const bar<U> &b) { /* ... */ }

template <typename T>
void foo<T>::baz() { /* ... */ }
```

Variadic Templates

- NEW in C++11: variadic templates.
 - Also called "parameter packs"
 - Allows templates to accept an arbitrary numbers of parameters and deal with them as a pack of types.
 - Expansion parameter pack (and variables of that type) done with ellipsis (...).
- Compiler performs pattern-matching when selecting which function or class template to use.
- See example on right.

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

// base case for template instantiation
template <typename T>
void print_list(const T &e) {
    std::cout << e << std::endl;
}

// recursive case for instantiation
template <typename T, typename ...Rest>
void print_list(const T &e, const Rest& ...rest) {
    std::cout << e << ' ';
    print_list(rest...);
}

int main() {
    // signature of the initial call:
    // print_list<int, char, double, std::string>
    // compiler will recursively instantiate
    // print_list until the base is reached.

    // instantiations:
    // print_list<int, [char, double, std::string]>
    // print_list<char, [double, std::string]>
    // print_list<double, [std::string]>
    // print_list<std::string> (no leftover type params)
    print_list(1, 'a', 3.14, std::string{"hi"});
} // output: 1 a 3.14 hi
```

sizeof...

- It is possible to get the number of elements in a variadic template's parameter pack.
- This is done with the `sizeof...` operator.
 - This returns a `constexpr std::size_t`.
- No static `typeof...` operator, however.
 - Only way to get the types of a parameter pack is to expand it in a template.

```
template <typename T, int N>
struct array { T elems[N]; }

// CTAD guide for array using variadic templates.
// Note the "+ 1" for the N parameter.
// This is because the initial T should also be
// counted as part of the array size.
template <typename T, typename ...Ts>
array(T, Ts...) -> array<T, sizeof...(Ts) + 1>;

int main() {
    // If only there was a standard library type
    // that achieved the same goal as this...
    // (std::array, perhaps?)
    array arr = {1, 2, 3};
}

// It is as-if sizeof... is implemented like below
// std::size_t sizeof...() { return 0; }
// template <typename T, typename ...Ts>
// std::size_t sizeof...(const T&, const T& ...ts) {
//     return 1 + sizeof...(ts...);
// }
```

Fold Expressions

- NEW in C++17: fold expressions.
- Use a parameter pack in expressions containing unary or binary operators to perform a left or right fold.
 - Parentheses around the fold expression are mandatory.
 - For the binary operator case:
 - The operators must be the same.
 - Optionally can take an initial value.
 - Can be used to replace some recursive function templates.

```
#include <iostream>

// Pre-C++17
template <typename T>
auto sum(T t1) { return t1 + 0; }

template <typename T, typename ...Ts>
auto sum(T t, Ts ...ts) { return t + sum(ts...); }

// Post-C++17

template <typename ...Ts>
auto sum_binary(Ts ...ts) {
    return (ts + ... + 6765); // 6765 is the initial value
}

template <typename ...Ts>
auto sum_unary(Ts ...ts) {
    return (... + ts); // no initial value
}

int main() {
    std::cout << sum_binary(3, 1, 2) << std::endl;
    std::cout << sum_unary(89, 96, 02) << std::endl;
}
// Output:
// 6771
// 187
```

Partial & Explicit Specialisation

The templates we've defined so far are completely generic.

- There are two ways we can refine our generic, primary templates for something more specific:
- Partial specialisation:
 - Refining ("specialising") the primary template to work with a subset of types.
 - T^*
 - `std::vector`

Explicit specialisation:

Refining the template for a specific, non-generic type.

- `std::string`
- `int`

Note: not all the template varieties can be partially specialised.

- Notably, function templates cannot be partially specialised.

When to Specialise

- You need to preserve existing semantics for something that would not otherwise work.
 - `std::is_pointer` is partially specialised over pointers.
- You want to write a type trait (coming next week).
 - `std::is_integral` is fully specialised for `int`, `long`, etc.
- There is an optimisation you can make for a specific type or family of types.
 - `std::vector<bool>` is fully specialised to reduce memory footprint.

When **NOT** to Specialise

- **Don't specialise function templates**
 - A function template cannot be partially specialised.
 - Fully specialised function templates are better done with overloads.
 - Herb Sutter has an article on this
 - <http://www.gotw.ca/publications/mill17.htm>
- You think it would be cool if you changed some feature of the class for a specific type.
 - People assume a class works the same for all types.
 - **Don't violate assumptions!**

(Not) Specialising Function Templates

- Though function templates cannot be partially specialised, they can be explicitly specialised.
- This creates incredibly confusing bugs.
- Do **NOT** specialise function templates.
- Use overloads instead.

```
#include <iostream>

template <typename T>
/* A */ void foo(T) { std::cout << "A\n"; }

template <>
/* B */ void foo(int *) { std::cout << "B\n"; }

template <typename T>
/* C */ void foo(T *) { std::cout << "C\n"; }

/* D */ void foo(int *) { std::cout << "D\n"; }

int main() {
    int p = 0;
    foo(&p); // which of A, B, C, D is used?
}

// Answer: D
```

(Not) Specialising Function Templates

- Though function templates cannot be partially specialised, they can be explicitly specialised.
- This creates incredibly confusing bugs.
- Do **NOT** specialised function templates.
- Use overloads instead.

```
#include <iostream>

template <typename T>
/* A */ void foo(T) { std::cout << "A\n"; }

template <>
/* B */ void foo(int *) { std::cout << "B\n"; }

template <typename T>
/* C */ void foo(T *) { std::cout << "C\n"; }

int main() {
    int p = 0;
    foo(&p); // which of A, B, C is used?
}

// Answer: C!!!

// The compiler considers only primary templates
// when deciding when if it should instantiate
// a function template. Once it has selected the
// primary template, only then will it look for any
// specialisations.
// Here, (C) is a better match than (A), so (C) is used
```

(Not) Specialising Function Templates

- Though function templates cannot be partially specialised, they can be explicitly specialised.
- This creates incredibly confusing bugs.
- Do **NOT** specialised function templates.
- Use overloads instead.

```
#include <iostream>

template <typename T>
/* A */ void foo(T) { std::cout << "A\n"; }

template <typename T>
/* C */ void foo(T *) { std::cout << "C\n"; }

template <>
/* B */ void foo(int *) { std::cout << "B\n"; }

int main() {
    int p = 0;
    foo(&p); // which of A, B, C is used?
}

// Answer: B!!!

// The compiler finds (C) as being the better primary
// template for the call to foo.
// Once it finds (C), it checks to see if there are
// any specialisations. Here, since (B) is declared after
// C, the compiler thinks it is a specialisation of (C),
// and so it will select (B) to call.
```

Partial Specialisation of Class Template

- You can partially specialise class types.
 - You cannot partially specialise a particular method of a class in isolation, however.
 - Partial specialisation of classes is particularly useful when writing type traits.
 - Compiler performs pattern matching on the given template arguments and the expected parameter.
 - If the primary template expects a `T` and the partial specialisation expects a `T*` and an `int*` is given, the compiler will select the partial specialisation since `T*` "matches" `int*` better than `T`.

```
template <typename T>
struct is_a_pointer {
    static constexpr auto value = false;
};

template <typename T>
struct is_a_pointer<T*> {
    static constexpr auto value = true;
};

// An example of a simple type trait.
// Starting generically, we assume any and every
// generic type is not a pointer.
// So, the answer to the question "is T a pointer?"
// is no (false).
// Through partial specialisation over pointer types,
// we can refine our answer to yes! (true)

// answer: false
constexpr auto int_is_pointer = is_a_pointer<int>::value;

// answer: true
constexpr auto ptr_is_pointer = is_a_pointer<void*>::value;
```

Explicit Specialisation of Class Template

- Explicit specialisation should only be done on class and variable templates.
- std::vector<bool> is an interesting example and here too.
 - Surprisingly,
std::vector<bool>::reference is not a bool&.
- In addition to the primary template, create a fully specialised version for a specific parameter (or set of parameters) for a template.

```
#include <iostream>

template <typename T>
struct is_void {
    static constexpr auto value = false;
};

template<>
struct is_void<void> {
    static constexpr auto value = true;
};

// The answer to the question:
// "Is this type void?"
// in general is "no".
// However, for void (and only void!), 
// the answer to: "is this type void?"
// is unambiguously yes.
// So, encode that information using
// explicit specialisation.
int main() {
    std::cout << std::boolalpha <<
        is_void<int>::value << ' ' <<
        is_void<void>::value << std::endl;
}
// output: false true
```

(Not) Specialising Alias Templates

- Alias template cannot be partially specialised.
- Alias templates cannot be explicitly specialised.
 - That would just be a regular alias!

Specialising Variable Templates

- Variable templates can be partially or fully specialised.
- Not many use cases of specialising variable templates.

```
#include <iostream>

// Actually, the mathematical constants
// are specified in the standard like this:
namespace std::numbers {
    template <typename T>
    constexpr T pi = /* unspecified */;

    template <>
    constexpr auto pi<double> = double(3.1415926535897932385L);

    template <>
    constexpr auto pi<float> = float(3.1415926535897932385);
}

// The reason for this is to disallow instantiating pi
// with an arbitrary type and losing precision.
// As above, only the provided explicit specialisations of
// pi are allowed to be used and trying to specialise or
// instantiate pi outside of this set is unspecified behaviour.
```

Implicit Instantiation

- We know the compiler instantiates templates.
 - But when exactly does it do it?
- The compiler **implicitly instantiates** a template only at its first point of use.
 - Thus, if you never use a template, it is never instantiated.
 - Further uses of the template used the cached instantiation.

```
template <typename T>
bool is_less_than(T t1, T t2) {
    return t1 < t2;
}

template <typename T>
    bool is_greater_than(T t1, T t2) {
return t1 > t2;
}

int main() {
    // first use of is_less_than<int, int>
    // this template is implicitly instantiated
    is_less_than(1, 4);

    // second use of is_less_than<int, int>
    // previous instantiation is used
    is_less_than(2, 5);

    // first use of is_less_than<char, char>
    // the template is implicitly instantiated
    is_less_than(2,2);
}
// no use of is_greater_than?
// no instantiations of it.
```

Inclusion Compilation Model

- When it comes to templates, we implement them in header files.
 - This is because template definitions need to be known at **compile time**.
- Will expose implementation details in the header file.
- Can cause slowdown in compilation as every file using the header file will have to instantiate the template.
 - It's then up the linker to ensure there is only 1 instantiation.
 - A lot of generated code is simply thrown away.

```
// in min.h
template <typename T>
T min(T t1, T t2) {
    return t1 < t2 ? t1 : t2;
}

// in me.cpp
#include "min.h"
void foo() {
    // complex calculation...
    auto m = min(var1, var2);
    // more complex calculations...
}

// in you.cpp
#include "min.h"
void bar() {
    // simple calculation
    auto mm = min(var0, var3);
    // more "simple" calculations...
}

// The function template min() was included in two
// different .cpp files. This means the compiler had
// to parse, instantiate, and store the same template
// twice, only for the linker to throw one version away...
```

Explicit Instantiation

- Sometimes, we want explicit control of when the compiler instantiates a template.
 - But should be avoided if it can be.
- This can alleviate some of the performance costs of using templates since the compiler only instantiates the template once.
 - It still has to parse the template definition, though.
- We can tell the compiler to only instantiate a template once and link the generated code after compilation.
 - This is especially useful for common instantiations of a template, such as `std::vector<int>`.

```
// in vec.h
template <typename T, int N>
struct vec {
    // other, very mathematical, implementation
    T elems[N];
};

// extern says that the template instantiation
// is defined in another .cpp file.
extern template struct vec<double, 3>;
extern template struct vec<double, 4>;

// in vec.cpp

// explicitly instantiate the extern templates.
// other .cpp files will use the generated code
// from this translation unit.
template struct vec<double, 3>;
template struct vec<double, 4>;
```

Two-Phase Translation

- Compiler processes each template in two phases:
 1. When compiler reaches the definition.
 - Happens once for each template for each translation unit.
 2. When compiler instantiates the template.
 - Happens once for each combination of template parameters.
- Error messages vary based on which phase the error was detected.
 - For **syntactical** issues, the error is reported when the compiler reaches the template definition.
 - For **semantic** (type-related) issues, the error is reported when the compiler instantiates the template.
- This has the benefit of reducing compiler work when a template is not used, but can lead to **dependent scope ambiguity**.

Dependent Scope Ambiguity

Consider example() on the right.

Question: what is this?

- If bar is a value, then this code is a multiplication.
- If bar is a type, this is a variable definition.
- If bar is a template, this code is ill-formed.

Since bar depends on the template parameter, which is not known until instantiation, we have to tell the compiler what we expect bar to be:

- By default, the compiler expects bar to be a value (nothing to do).
- If bar is a type, we need to prepend typename before using bar.
- If bar is a template, we need to prepend template before using bar.

```
int a;

template <typename T>
void example() {
    T::bar * a; // what is this?
}

template <typename T>
void example_value() {
    // this is a multiplication of
    // T::bar and the global variable a
    T::bar * a;
}

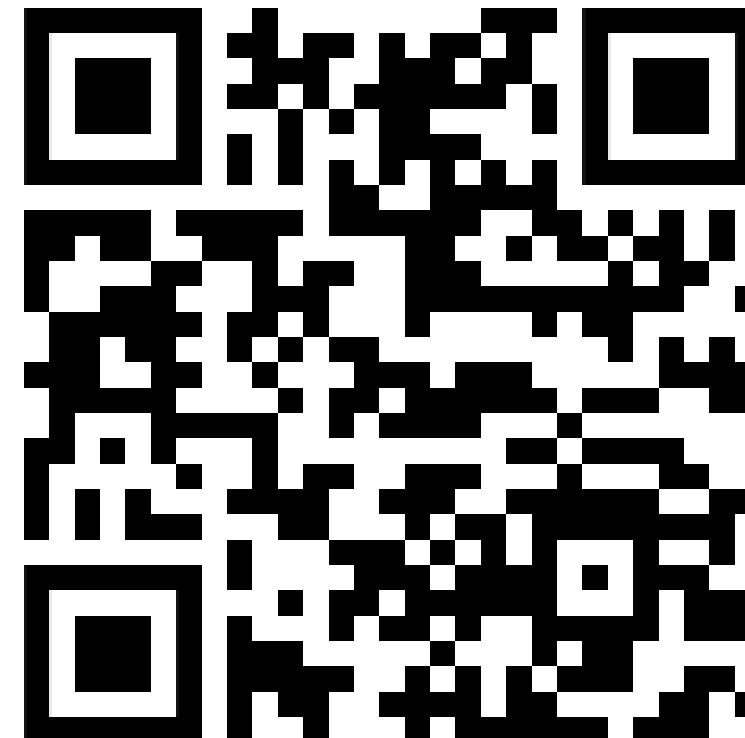
template <typename T>
void example_type() {
    // this is definition of a local variable a
    // (a pointer to the type T::bar)
    typename T::bar * a;
}

template <typename T>
void example_template() {
    T:: template bar * a; // this code is ill-formed.
}
```

Considerations for Templates

- C++ templates are an extremely powerful way to do generic programming.
 - Fast, efficient, and *automatic*.
- Tenet of C++: don't pay for what you don't use.
 - Use of templates has a cost.
 - Overuse of templates in large projects absolutely cripples compilation times.
- Some guidelines when using templates:
 - For general datastructures and generic, relatively small, and modular algorithms, template use is completely fine.
 - Always prefer functions, function overloads, and real class-types over templates unless you have a reason to generalise.
 - YAGNI still applies – Ya Ain't Gonna Need It!
- If compilation time is too long, consider using explicit instantiation.

Feedback (stop recording)



COMP6771

Advanced C++ Programming

9.1 Metaprogramming

In this Lecture

What?

- Template Metaprogramming
- `constexpr` pt. 2
- Perfect forwarding
- Concepts

Why?

- C++'s compile-programming feature set is unique.
- Many production codebases which use templates also use metaprogramming.

Metaprogramming in C++

- Metaprogramming:
 - "a programming technique in which computer programs have the ability to treat other programs as their data".
- Historically, templates were used to do metaprogramming in C++ (**Template Meta-Programming**).
 - Templates are [Turing complete](#).
 - C++ TMP uses template instantiation to drive compile-time evaluation.
- Modern C++ metaprogramming uses more than just TMP.
- Metaprogramming turns the compiler into a code generator:

Template Metaprogramming (TMP)

- TMP is built off of **SFINAE** and **partial/explicit specialisation**.
- Substitution Failure Is Not An Error (SFINAE) is used to manage overload sets.
- Specialisation is used to implement type traits and type transformations.

```
// An example of using specialisation to implement
// a compile-time "if" that can be used where a type
// is expected (such as when making a type alias).

// Primary template: assume B is "true"
template <bool B, typename T, typename F>
struct if_t { using type = T; }

// Partial Specialisation: used when B is "false"
template <typename T, typename F>
struct if_t<false, T, F> { using type = F; };

int main() {
    using int16_t = if_t<sizeof(int) == 2, int, short>::type;
    return int16_t{42};
}
```

SFINAE: Function Templates

Substitution Failure Is Not An Error

- If substitution of a template argument (whether given or deduced) into a template parameter causes a function template to be ill-formed, it is **silently discarded** and is *not* a compile error!
- Can be used to "remove" function templates from being considered during overload resolution.

```
#include <type_traits>

// When "B" is true, maybe::type exists and is T.
// When "B" is false... there is no type!
template <bool B, typename T> struct maybe { using type = T; };
template <typename T> struct maybe<false, T> {};
template <bool B, typename T>
using maybe_t = typename maybe<B, T>::type;

template <typename T>
// For non-integral types, there is no return type.
// That is ill-formed, so this overload is silently discarded
maybe_t<std::is_integral_v<T>, T>
secret_algorithm(T i) { return i * 2; }

template <typename T>
// For non-floating point types, there is no return type.
// That is ill-formed, so this overload is silently discarded
maybe_t<std::is_floating_point_v<T>, T>
secret_algorithm(T fp) { return fp * T{3.14}; }

int main() {
    int secret_int = secret_algorithm(6771);
    float secret_float = secret_algorithm(3.14f);
}
// Here, we don't have two overloads of "secret_algorithm";
// For "int" vs. "float", we explicitly remove the other
// overload from the resolution set using SFINAE!
```

SFINAE: Types

Substitution Failure Is Not An Error

- If a template parameter is used to form a type, or access a member of type, and that use would cause the code to be ill-formed, that code is *silently discarded*.
- This happens when:
 - Attempting to form an array with length 0, or array to void, or any invalid array.
 - Attempting to use a member of a type in an inappropriate context (e.g., using a non-type where a type is required).
 - Etc. Full list of cases [here](#).

```
template <typename T>
class is_class {
    using yes = char[3];
    using no = char[2];

    template <typename C>
    // selected if C is a class type.
    // this is because this member template
    // parameter accepts a pointer to a member function.
    // this is only valid for class-types.
    static yes &test(int C::*);
```

// if C is not a class type,
// this overload is selected.

```
static no &test(...);
```

```
public:
    // the "test" is to see if "T" is a class-type.
    // if it is, then the return type of "test()" will be
    // yes. we can check the size of the return type to
    // determine which overload was selected.
```

```
static constexpr auto value =
    sizeof(test<T>(nullptr)) == sizeof(yes);
};
```

SFINAE: Expressions

Substitution Failure Is Not An Error

- NEW in C++11: expression SFINAE.
 - The following expression errors are SFINAE errors.
 - Ill-formed expressions used in a template parameter type.
 - Ill-formed expressions used in a function type.
 - These are not errors but are *silently discarded*.

```
template<int I>
void ex(char(&array1)[I % 2 == 0]) {} /* @ */

template<int I>
void ex(char(&array2)[I % 2 == 1]) {} /* $ */

// When "I" is even, (@) is selected.
// When "I" is odd, ($) is selected.
//
// This is because arrays of length 0 are ill-formed.
// When "I" is even, "I % 2 == 0" is true
//   -> this boolean is converted to int{1}.
//   -> "array1"'s type becomes char(&)[1].
//   whereas "array2"'s type becomes char(&)[0]
// which is ill-formed.
//
// Likewise, when "I" is odd, "I % 2 == 1" is true.
//   -> this boolean is converted to int{1}.
//   -> "array2"'s type becomes char(&)[1].
//   whereas "array1"'s type becomes char(&)[0].
```

Type Traits

- Type traits are a way to ask questions about a type at compile-time.
 - Is T a reference type?
 - What is the rank of T? (non-array is 0, otherwise the number of [] in the type)
- Implemented as:
 - A primary struct template that answers the question in general.
 - Specialisations that answer the question specifically.
 - Questions for type use a type member called `type`.
 - Boolean questions use a `static constexpr` boolean called `value`.
- Why?
 - *Extremely* useful for writing optimised generic code.
 - Allow explicit overload set management when writing function templates.

```
// Here are an assortment of some useful type traits.
```

```
template <typename T>
// in general, T is not a reference
struct is_reference {
    static constexpr auto value = false;
};

template <typename T>
struct is_reference<T&> { // but T& is!
    static constexpr auto value = true;
};

template <typename T>
struct is_reference<T&&> { // and so is T&&!
    static constexpr auto value = true;
};

template <typename T>
struct rank_of { // in general, T is rank 0.
    static constexpr auto value = 0;
};

template <typename T>
// but arrays have rank 1!
// T could have multiple []: recursively apply the type
// trait.
struct rank_of<T[]> {
    static constexpr auto value = 1 + rank_of<T>::value;
};
```

Type Transformations

- Type transformations allow modification of an existing type.
- Implemented similar to type traits.
- Can be used to generically add type modifiers:
 - such as `const`, `volatile`, `&`, `*`, etc.
- Can also be used to *remove* modifiers from a type.

```
// Some examples of basic type transformations

template <typename T>
// in general, assume T is not const.
// nothing to remove
struct rm_const { using type = T; };

template <typename T>
// but when T is const...
// Unshell the const from T
// and use the raw type as the result
struct rm_const<const T> { using type = T; };

template <typename T>
// can always add a & to a type (unless void).
struct add_lvalue { using type = T&; };

template <typename T>
// can always add a * to a type.
struct add_pointer { using type = T*; };
```

Standard Type Traits

- The Standard Library has many useful type traits in the [`<type_traits>`](#) header.
- Where possible, you should always try to use or reuse standard type traits.
 - When making your own type traits, it is generally a good idea to simply compose existing standard ones.

```
#include <type_traits>

template <typename T>
struct is_int_pointer {
    static constexpr auto value = false;
};

// We could do this...
template <typename T>
struct is_int_pointer {
    static constexpr auto value = false;
};

template <>
struct is_int_pointer<int *> {
    static constexpr auto value = true;
};

// but why do that when you have the Standard Library?
// Here, we inherit either from true_type or false_type
// depending on if "T" is an "int*" or not.
template <typename T>
struct is_int_pointer : std::conditional_t<
    std::is_same_v<T, int *>,
    std::true_type, // std::true_type::value == true.
    std::false_type // std::true_type::value == false.
>
{};
```

Shortened Type Traits

- NEW! In C++14, you can refer to the Standard Library's type-based traits through a shortened alias.
- NEW! In C++17, you can refer to the Standard Library's boolean-based traits through a shortened alias.

```
#include <type_traits>

// should be int
using return_type =
    std::invoke_result_t<int(char), char>;

// this replaces
// using return_type =
// typename std::invoke_result<int(char), char>::type;

// should be true.
constexpr bool was_it_an_int =
    std::is_same_v<return_type, int>;

// this replaces
// constexpr bool =
//     std::is_same<return_type, int>::value;
```

TMP: Just a Sample

- What has been shown is just a sample of what can be done with TMP.
- Depending on how clever you want to be, it can become much more difficult.
 - Most compiler bugs are found within its template machinery.
 - Some resources to see more examples of TMP:
 - Dr. Walter E. Brown's talk at CppCon 2014 ([part 1](#) & [part 2](#))
 - Jody Hagins talk on [type traits](#)
 - [An implementation of named tuples](#) through the (ab)use of templates in C++..

decltype

- Semantic equivalent of a "typeof" operator.
 - Type deduced from expression.
- **Rule 1:**
 - If expression **e** is any of:
 - variable in local scope
 - variable in namespace scope
 - static member variable
 - function parameters
 - then the result is of type T
- **Rule 2:** if **e** is an lvalue, result is T&.
- **Rule 3:** if **e** is an xvalue, result is T&&.
- **Rule 4:** if **e** is a prvalue, result is T.
 - xvalue/prvalue are forms of rvalues.
 - We do not require you to know this.
- Non-simplified set of rules can be found [here](#).

```
// Just a lowly const int
const int i = 0;

// const int - needs to be initialised
decltype(i) x = 0;

// int& - needs to be initialised
// parenthesised expressions always are lvalues
decltype((i)) j = i;

// What is the result of j + i? int!
// b is of type "int" (currently uninitialized)
decltype(j+i) b;

const int *ptr = nullptr;

// Result of *ptr is const int&
// hence c is const int& so must be initialized
decltype(*ptr) c = i;

// int - prvalue
decltype(5) z = 6771;
```

Determining Return Types (Problem)

- Ever wondered what the advantage is of declaring a function using auto syntax?
- Consider foo() to the right.
 - We want the return type to be the result type of $t + u$.
 - Using decltype, this may be achievable!
- Unfortunately, foo2() has issues too...
 - t and u are not in scope yet by the time they are needed!

```
template <typename T, typename U>
??? foo(const T &t, const U &u) {
    return t + u;
}
```

```
template <typename T, typename U>
auto foo2(const T &t, const U &u)-> decltype(t + u) {
    return t + u;
}
```

// ERROR! t and u have not been declared yet!

Determining Return Types (Solution)

- If the decltype statement was *after* the variable declarations, then it would be fine.
- This is what auto-function syntax allows us to do.
 - Once this was realised, the decltype return type became optional.
 - Usually, deduce the return type from the function body.
 - But if the deduced type is not what is required (e.g., an implicit conversion is desired), can fall back on decltype.

```
#include <iostream>

template <typename T, typename U>
// Here, instead of letting the compiler deduce
// the type of t + u (which should be "int")
// we say the return type should be the result
// of a narrowing conversion from t to double
// and then adding u to that double.
auto foo(T t, U u) -> decltype(double(t) + u) {
    return t + u;
}

int main() {
    std::cout << foo(42, 6771) << std::endl;
}
```

decltype(auto)

- C++ has two sets of deduction rules.
- auto deduction rules:
 - Same as template argument deduction rules.
 - Also called object rules, since auto never deduces references or top-level const.
- decltype rules:
 - Will deduce references and top-level const.
 - Parenthesised expressions (e.g. `decltype((x))`) return lvalue references.
- What if you want both?
 - Let the compiler deduce the return object type with auto...
 - But then preserve its value category with decltype!
 - Mainly used with function return types only.

```
#include <iostream>
#include <type_traits>

int a = 0;

int main() {
    // deduce the type of "a" as an int object
    // but it is being used as an lvalue
    // (note the parentheses)
    // so altogether: i is int&!
    decltype(auto) i = (a);

    // deduce the type of 3.14f as a float object.
    // it is a pure rvalue
    // (note: it is a float literal)
    // so altogether: j is just a float!
    decltype(auto) j = 3.14f;

    std::cout << std::boolalpha;
    std::cout << std::is_same_v<decltype(i), int&> << ' '
        << std::is_same_v<decltype(j), float> << '\n';
}

// Output: true true
```

Unevaluated Contexts

- C++ has *eager evaluation*.
 - In an expression, the operands are evaluated at runtime for their effects.
 - But the result type of the expression is already known at compile-time.
- An **unevaluated context** is when:
 - Operands are not evaluated -> no runtime effects.
 - Only the statically-known types of expressions are used to further calculate types.
- Some unevaluated contexts:
 - When using `decltype()`
 - When using `noexcept()`
 - `noexcept` is equivalent to `noexcept(true)`
 - When using `sizeof(variable)` rather than `sizeof(type)`
 - [Full list](#)

decltype & std::declval<T>

- A common metaprogramming pattern is using `std::declval` with `decltype`.
- `std::declval` is a function template *declaration* that:
 - Returns an rvalue reference to `T`.
 - Let's one use the member functions of `T` with `decltype` without needing to worry about constructors.
- `std::declval<T>` can only be used in unevaluated contexts.
 - Otherwise, a defintion would be required.

```
// How to test the return type of a method is correct
#include <iostream>
#include <type_traits>
#include <utility>

class Integer {
public:
    /* Other implementation */
    const int &i() const;
private:
    int i_;
};

int main() {
    auto is_correct = std::is_same_v<decltype(
        // use std::declval to "get" a const Integer&&
        // then, we can inspect its methods in decltype!
        std::declval<const Integer>().i()
    ), const int&>;
    std::cout << std::boolalpha << is_correct << "\n";
}

// Output: true
```

constexpr Revisited

- We have already seen some uses of `constexpr` when defining compile-time calculable variables.
- Since C++11, the use cases of `constexpr` have grown:
 - `constexpr` variables
 - `constexpr if`
 - `constexpr` functions.
- Let's explore `constexpr` more in-depth.

constexpr Variables: Revision

- A `constexpr` variable is a variable whose value is calculable at compile-time.
- Unless there is code which requires the existence of it, a `constexpr` variable does not exist at runtime.
 - Its value is hardcoded into the final executable by the compiler.
- Provides benefits over macros for global constants.
 - Is scoped and code will not compile if there is a name conflict.
 - Is type-checked by the compiler.

```
constexpr int N = 4;

int get_int(); // defined elsewhere

int main() {
    const int M = get_int();

    // not OK: M not known until runtime
    int arr1[M] = {0};

    // OK: N is a constexpr variable
    int arr2[N] = {0};
}
```

if constexpr

Read "constexpr if", written as "if constexpr"

- NEW! in C++17: a compile-time if-statement.
 - Can be used anywhere a regular if-statement is usable.
 - The condition is evaluated at compile-time and only **one** of the branches is compiled into the final binary.
 - The other branch is discarded.
 - Both branches are checked for correct syntax, but only the taken branch is checked for semantics errors.

```
#include <type_traits>

template <typename T>
auto value_or_deref(const T t) {
    // Check if T is a pointer.
    // If it is, dereference t.
    if constexpr (std::is_pointer_v<T>) {
        return *t;
    } else {
        // otherwise, return t directly.
        // For non-pointers, only this branch
        // appears in the final executable!
        return t;
    }
}

template <typename T>
auto compile_error(T) {
    // It is illegal to dereference non-pointers.
    // This kind of error is caught regardless of
    // which branch would be taken at compile-time.
    if constexpr (sizeof(T) == 4) {
        int i = 0;
        return *i;
    } else {
        float f = 0;
        return *f;
    }
}
```

constexpr Functions

- A `constexpr` function is a function that *may* be called and computed at compile-time.
 - Happens when the arguments to the function are also `constexpr`.
- A `constexpr` function body can contain:
 - Variable definitions calculable at compile-time.
 - Loops.
 - Calls to other `constexpr` functions.
 - Calculations involving variables calculable at compile-time.
 - Complete list [here](#).
 - If any of the above conditions are not met, the function will be called at runtime.
- The return value of a `constexpr` function can be used to initialise `constexpr` variables.

```
int get_int();

constexpr int factorial(int n) {
    int res = 1;
    while (n > 1) res *= n--;
    return res;
}

int main() {
    // Will be calculated at compile-time because
    // all arguments to factorial are constexpr.
    int arr1[factorial(1)] = {factorial(4)};

    // Error: get_int() is not constexpr
    // so factorial will be called at runtime.
    // cannot initialise a constexpr variable
    // with the return value from a function
    // that is called at runtime.
    constexpr int i = factorial(get_int());
}
```

consteval Functions

- NEW! in C++20: consteval functions.
- Same rules as a constexpr function, except it is guaranteed to be called at compile-time.

```
int get_int();

consteval int factorial(int n) {
    int res = 1;
    while (n > 1) res *= n--;
    return res;
}

int main() {
    // Guaranteed to be called at compile time.
    int arr1[factorial(1)] = {factorial(4)};

    // Error: get_int() cannot be used as an argument
    // to a consteval function because it is not
    // a constant expression.
    constexpr int i = factorial(get_int());
}
```

Literal Types

- A literal type is a type that is initialisable at compile-time.
 - All fundamental types are literal types.
- User-defined types can also be literal types.
 - You must define at least one `constexpr/consteval` constructor.
 - Any base classes must also be literal.
 - Virtual base classes are not allowed.
 - Literal types can also be used in `constexpr/consteval` functions.

```
#include <cmath>
#include <iostream>

class point2d {
public:
    constexpr point2d() noexcept = default;
    constexpr point2d(double x, double y) noexcept
        : x_{x}, y_{y} {}

    constexpr double x() const noexcept { return x_; }
    constexpr double y() const noexcept { return y_; }

private:
    double x_;
    double y_;
};

constexpr
double distance(const point2d &p, const point2d &q) {
    return std::sqrt(p.x() * q.x() + p.y() * q.y());
}

int main() {
    // Will print 5 -- value calculated at compile-time.
    std::cout << distance({0, 3}, {4, 0}) << std::endl;
}
```

Compile-time Calculation: Templates vs. `constexpr` Functions

- Template Metaprogramming:
 - More powerful / can do more things than `constexpr` functions.
 - Much harder to read and reason about if there's a bug.
 - Can cause code explosion due to one-time use template instantiations.
- `constexpr` functions
 - Can use `if constexpr` and other familiar programming syntax.
 - Much easier to debug and reason about.
 - Newer – can do more and more things with `constexpr` functions with each Standard
 - Since C++20: `try/catch`, some dynamic memory allocation at compile-time.
- General guidelines.
 - Prefer `constexpr` functions if they can be used. Readability is important!
 - Fallback onto TMP if ultimate power is desired.
 - As always, try to use the right tool for the job.

Perfect Forwarding

- Often when programming you'll want to wrap a function call to perform some extra logic.
- This presents a problem:
 - How should we accept the parameters to pass onto the underlying function? Does it make sense to use the same types as the function we are wrapping?
 - How to avoid needless copying? Should we const T&? What about T&&?
- **Perfect Forwarding** is a way to wrap a function and pass it its arguments *perfectly*.
- This relies on C++'s binding rules and Reference Collapsing.

Binding (Non-templates)

Binding table for a concrete T (int, char, etc.)	Arguments				
Parameters		lvalue	const lvalue	rvalue	const rvalue
	T	✓	✓	✓	✓
	T&	✓	✗	✗	✗
	const T&	✓	✓	✓	✓
	T&&	✗	✗	✓	✗

T and const T& bind to everything!

- Unfortunately, T creates a copy
- const T& is immutable – what if we need to modify the value?

The rules change when templates are involved.

Binding (Templates)

Binding table for typename T	Arguments				
		lvalue	const lvalue	rvalue	const rvalue
Parameters	T	✓	✓	✓	✓
	T&	✓	✗	✗	✗
	const T&	✓	✓	✓	✓
	T&&	✓	✓	✓	✓

When T is a template parameter, T&& binds to everything!

- Binds even if T is deduced to be const or not.
- Binds even if T is a value, an lvalue reference, or an rvalue reference.

Reference Collapsing

- **Question:** what is decltype(t) in the calls to accept_everything?
 - Call (1): int&
 - Call (2): const int && &!?
 - Call (3): int && &&!?
- C++ has *reference collapsing* rules:
 - T& & -> T&
 - an lvalue to an lvalue is still an lvalue.
 - T&& & -> T&
 - an lvalue to an rvalue is still an lvalue.
 - T& && -> T&
 - an rvalue to an lvalue is still an lvalue.
 - T&& && -> T&&
 - an rvalue to an rvalue is still an rvalue.
- Through reference collapsing, the value category of an argument is preserved across function calls.

```
template <typename T>
void accept_everything(T &&t);

int main() {
    int i = 0;
    const int &j = i;
    int &&k = 6771;

    accept_everything(i); // 1
    accept_everything(j); // 2
    accept_everything(k); // 3
}
```

Forwarding References

- Due to T&& binding to everything and reference collapsing, T&& is known as a *forwarding reference*.
- Called a "universal reference" in older (circa 2011) texts.
- Used mainly in function templates that act as wrappers to pass arguments along to wrapped functions.
- To actually forward the argument, use std::forward

```
// This might be how std::make_unique is implemented
#include <memory>

struct foo { int a; bool b; const char *c; };

template <typename T, typename ...Args>
std::unique_ptr<T> my_make_unique(
    // note the Args&& - this is a forwarding reference!
    Args&& ...args
) {
    // We are going to construct a T from the arguments
    // that we got passed from the caller.
    // We will pass the arguments to T's constructor
    // exactly the way they were passed to us!
    T *ptr = new T{
        // we call std::forward to actually the arguments along.
        // This will call std::forward on each argument
        // inside the parameter pack
        std::forward<Args>(args)...
    };
    return std::unique_ptr{ptr};
}

auto a = 3;
auto msg = "hi!";

// my_make_unique: T = foo, Args = [int&, bool, const char * &&]
//
// foo's constructor will be called as if we called it directly
// with arguments of these types.
auto up = my_make_unique<foo>(a, a == 3, std::move(msg));
```

Uses of std::forward

The only real use for std::forward is when you want to wrap a function with a parameterised type. This could be because:

- You want to do something else before or after.
 - std::make_unique / std::make_shared need to wrap a std::unique / std::shared_ptr variable.
 - A benchmarking library might wrap a function call with timers.
- You want to do something slightly different.
 - std::vector::emplace() constructs its element type in uninitialised memory.
- You want to add an extra parameter.
 - E.g. always call a function with the last parameter as 1.
 - This isn't usually very useful, because it can be achieved with std::bind or lambda functions more easily.

A Major Problem with Templates

- Recall that templates have two-phase translation.
- The template definition is checked for syntax errors only.
- Upon instantiation, the definition is fully type-checked.
- This leads to type errors being detected extremely late during the instantiation phase.
 - C++ is **infamous** for *terrible* template error messages.
 - Prior to C++20 there was no way to specify to the compiler requirements of types expected to be used with a template.

```
[line 1] In file included from /usr/include/c++/11/set.h:  
[line 2] 10 #include <utility>  
[line 3] 11 #include <functional>  
[line 4] 12 #include <memory>  
[line 5] 13 #include <new>  
[line 6] 14 #include <typeinfo>  
[line 7] 15 #include <assert>  
[line 8] 16 #include <cmath>  
[line 9] 17 #include <limits>  
[line 10] 18 #include <iomanip>  
[line 11] 19 #include <iostream>  
[line 12] 20 #include <iosfwd>  
[line 13] 21 #include <ios>  
[line 14] 22 #include <iterator>  
[line 15] 23 #include <map>  
[line 16] 24 #include <memory>  
[line 17] 25 #include <new>  
[line 18] 26 #include <optional>  
[line 19] 27 #include <random>  
[line 20] 28 #include <ratio>  
[line 21] 29 #include <string>  
[line 22] 30 #include <type_traits>  
[line 23] 31 #include <utility>  
[line 24] 32  
[line 25] 33  
[line 26] 34  
[line 27] 35  
[line 28] 36  
[line 29] 37  
[line 30] 38  
[line 31] 39  
[line 32] 40  
[line 33] 41  
[line 34] 42  
[line 35] 43  
[line 36] 44  
[line 37] 45  
[line 38] 46  
[line 39] 47  
[line 40] 48  
[line 41] 49  
[line 42] 50  
[line 43] 51  
[line 44] 52  
[line 45] 53  
[line 46] 54  
[line 47] 55  
[line 48] 56  
[line 49] 57  
[line 50] 58  
[line 51] 59  
[line 52] 60  
[line 53] 61  
[line 54] 62  
[line 55] 63  
[line 56] 64  
[line 57] 65  
[line 58] 66  
[line 59] 67  
[line 60] 68  
[line 61] 69  
[line 62] 70  
[line 63] 71  
[line 64] 72  
[line 65] 73  
[line 66] 74  
[line 67] 75  
[line 68] 76  
[line 69] 77  
[line 70] 78  
[line 71] 79  
[line 72] 80  
[line 73] 81  
[line 74] 82  
[line 75] 83  
[line 76] 84  
[line 77] 85  
[line 78] 86  
[line 79] 87  
[line 80] 88  
[line 81] 89  
[line 82] 90  
[line 83] 91  
[line 84] 92  
[line 85] 93  
[line 86] 94  
[line 87] 95  
[line 88] 96  
[line 89] 97  
[line 90] 98  
[line 91] 99  
[line 92] 100  
[line 93] 101  
[line 94] 102  
[line 95] 103  
[line 96] 104  
[line 97] 105  
[line 98] 106  
[line 99] 107  
[line 100] 108  
[line 101] 109  
[line 102] 110  
[line 103] 111  
[line 104] 112  
[line 105] 113  
[line 106] 114  
[line 107] 115  
[line 108] 116  
[line 109] 117  
[line 110] 118  
[line 111] 119  
[line 112] 120  
[line 113] 121  
[line 114] 122  
[line 115] 123  
[line 116] 124  
[line 117] 125  
[line 118] 126  
[line 119] 127  
[line 120] 128  
[line 121] 129  
[line 122] 130  
[line 123] 131  
[line 124] 132  
[line 125] 133  
[line 126] 134  
[line 127] 135  
[line 128] 136  
[line 129] 137  
[line 130] 138  
[line 131] 139  
[line 132] 140  
[line 133] 141  
[line 134] 142  
[line 135] 143  
[line 136] 144  
[line 137] 145  
[line 138] 146  
[line 139] 147  
[line 140] 148  
[line 141] 149  
[line 142] 150  
[line 143] 151  
[line 144] 152  
[line 145] 153  
[line 146] 154  
[line 147] 155  
[line 148] 156  
[line 149] 157  
[line 150] 158  
[line 151] 159  
[line 152] 160  
[line 153] 161  
[line 154] 162  
[line 155] 163  
[line 156] 164  
[line 157] 165  
[line 158] 166  
[line 159] 167  
[line 160] 168  
[line 161] 169  
[line 162] 170  
[line 163] 171  
[line 164] 172  
[line 165] 173  
[line 166] 174  
[line 167] 175  
[line 168] 176  
[line 169] 177  
[line 170] 178  
[line 171] 179  
[line 172] 180  
[line 173] 181  
[line 174] 182  
[line 175] 183  
[line 176] 184  
[line 177] 185  
[line 178] 186  
[line 179] 187  
[line 180] 188  
[line 181] 189  
[line 182] 190  
[line 183] 191  
[line 184] 192  
[line 185] 193  
[line 186] 194  
[line 187] 195  
[line 188] 196  
[line 189] 197  
[line 190] 198  
[line 191] 199  
[line 192] 200  
[line 193] 201  
[line 194] 202  
[line 195] 203  
[line 196] 204  
[line 197] 205  
[line 198] 206  
[line 199] 207  
[line 200] 208  
[line 201] 209  
[line 202] 210  
[line 203] 211  
[line 204] 212  
[line 205] 213  
[line 206] 214  
[line 207] 215  
[line 208] 216  
[line 209] 217  
[line 210] 218  
[line 211] 219  
[line 212] 220  
[line 213] 221  
[line 214] 222  
[line 215] 223  
[line 216] 224  
[line 217] 225  
[line 218] 226  
[line 219] 227  
[line 220] 228  
[line 221] 229  
[line 222] 230  
[line 223] 231  
[line 224] 232  
[line 225] 233  
[line 226] 234  
[line 227] 235  
[line 228] 236  
[line 229] 237  
[line 230] 238  
[line 231] 239  
[line 232] 240  
[line 233] 241  
[line 234] 242  
[line 235] 243  
[line 236] 244  
[line 237] 245  
[line 238] 246  
[line 239] 247  
[line 240] 248  
[line 241] 249  
[line 242] 250  
[line 243] 251  
[line 244] 252  
[line 245] 253  
[line 246] 254  
[line 247] 255  
[line 248] 256  
[line 249] 257  
[line 250] 258  
[line 251] 259  
[line 252] 260  
[line 253] 261  
[line 254] 262  
[line 255] 263  
[line 256] 264  
[line 257] 265  
[line 258] 266  
[line 259] 267  
[line 260] 268  
[line 261] 269  
[line 262] 270  
[line 263] 271  
[line 264] 272  
[line 265] 273  
[line 266] 274  
[line 267] 275  
[line 268] 276  
[line 269] 277  
[line 270] 278  
[line 271] 279  
[line 272] 280  
[line 273] 281  
[line 274] 282  
[line 275] 283  
[line 276] 284  
[line 277] 285  
[line 278] 286  
[line 279] 287  
[line 280] 288  
[line 281] 289  
[line 282] 290  
[line 283] 291  
[line 284] 292  
[line 285] 293  
[line 286] 294  
[line 287] 295  
[line 288] 296  
[line 289] 297  
[line 290] 298  
[line 291] 299  
[line 292] 300  
[line 293] 301  
[line 294] 302  
[line 295] 303  
[line 296] 304  
[line 297] 305  
[line 298] 306  
[line 299] 307  
[line 300] 308  
[line 301] 309  
[line 302] 310  
[line 303] 311  
[line 304] 312  
[line 305] 313  
[line 306] 314  
[line 307] 315  
[line 308] 316  
[line 309] 317  
[line 310] 318  
[line 311] 319  
[line 312] 320  
[line 313] 321  
[line 314] 322  
[line 315] 323  
[line 316] 324  
[line 317] 325  
[line 318] 326  
[line 319] 327  
[line 320] 328  
[line 321] 329  
[line 322] 330  
[line 323] 331  
[line 324] 332  
[line 325] 333  
[line 326] 334  
[line 327] 335  
[line 328] 336  
[line 329] 337  
[line 330] 338  
[line 331] 339  
[line 332] 340  
[line 333] 341  
[line 334] 342  
[line 335] 343  
[line 336] 344  
[line 337] 345  
[line 338] 346  
[line 339] 347  
[line 340] 348  
[line 341] 349  
[line 342] 350  
[line 343] 351  
[line 344] 352  
[line 345] 353  
[line 346] 354  
[line 347] 355  
[line 348] 356  
[line 349] 357  
[line 350] 358  
[line 351] 359  
[line 352] 360  
[line 353] 361  
[line 354] 362  
[line 355] 363  
[line 356] 364  
[line 357] 365  
[line 358] 366  
[line 359] 367  
[line 360] 368  
[line 361] 369  
[line 362] 370  
[line 363] 371  
[line 364] 372  
[line 365] 373  
[line 366] 374  
[line 367] 375  
[line 368] 376  
[line 369] 377  
[line 370] 378  
[line 371] 379  
[line 372] 380  
[line 373] 381  
[line 374] 382  
[line 375] 383  
[line 376] 384  
[line 377] 385  
[line 378] 386  
[line 379] 387  
[line 380] 388  
[line 381] 389  
[line 382] 390  
[line 383] 391  
[line 384] 392  
[line 385] 393  
[line 386] 394  
[line 387] 395  
[line 388] 396  
[line 389] 397  
[line 390] 398  
[line 391] 399  
[line 392] 400  
[line 393] 401  
[line 394] 402  
[line 395] 403  
[line 396] 404  
[line 397] 405  
[line 398] 406  
[line 399] 407  
[line 400] 408  
[line 401] 409  
[line 402] 410  
[line 403] 411  
[line 404] 412  
[line 405] 413  
[line 406] 414  
[line 407] 415  
[line 408] 416  
[line 409] 417  
[line 410] 418  
[line 411] 419  
[line 412] 420  
[line 413] 421  
[line 414] 422  
[line 415] 423  
[line 416] 424  
[line 417] 425  
[line 418] 426  
[line 419] 427  
[line 420] 428  
[line 421] 429  
[line 422] 430  
[line 423] 431  
[line 424] 432  
[line 425] 433  
[line 426] 434  
[line 427] 435  
[line 428] 436  
[line 429] 437  
[line 430] 438  
[line 431] 439  
[line 432] 440  
[line 433] 441  
[line 434] 442  
[line 435] 443  
[line 436] 444  
[line 437] 445  
[line 438] 446  
[line 439] 447  
[line 440] 448  
[line 441] 449  
[line 442] 450  
[line 443] 451  
[line 444] 452  
[line 445] 453  
[line 446] 454  
[line 447] 455  
[line 448] 456  
[line 449] 457  
[line 450] 458  
[line 451] 459  
[line 452] 460  
[line 453] 461  
[line 454] 462  
[line 455] 463  
[line 456] 464  
[line 457] 465  
[line 458] 466  
[line 459] 467  
[line 460] 468  
[line 461] 469  
[line 462] 470  
[line 463] 471  
[line 464] 472  
[line 465] 473  
[line 466] 474  
[line 467] 475  
[line 468] 476  
[line 469] 477  
[line 470] 478  
[line 471] 479  
[line 472] 480  
[line 473] 481  
[line 474] 482  
[line 475] 483  
[line 476] 484  
[line 477] 485  
[line 478] 486  
[line 479] 487  
[line 480] 488  
[line 481] 489  
[line 482] 490  
[line 483] 491  
[line 484] 492  
[line 485] 493  
[line 486] 494  
[line 487] 495  
[line 488] 496  
[line 489] 497  
[line 490] 498  
[line 491] 499  
[line 492] 500  
[line 493] 501  
[line 494] 502  
[line 495] 503  
[line 496] 504  
[line 497] 505  
[line 498] 506  
[line 499] 507  
[line 500] 508  
[line 501] 509  
[line 502] 510  
[line 503] 511  
[line 504] 512  
[line 505] 513  
[line 506] 514  
[line 507] 515  
[line 508] 516  
[line 509] 517  
[line 510] 518  
[line 511] 519  
[line 512] 520  
[line 513] 521  
[line 514] 522  
[line 515] 523  
[line 516] 524  
[line 517] 525  
[line 518] 526  
[line 519] 527  
[line 520] 528  
[line 521] 529  
[line 522] 530  
[line 523] 531  
[line 524] 532  
[line 525] 533  
[line 526] 534  
[line 527] 535  
[line 528] 536  
[line 529] 537  
[line 530] 538  
[line 531] 539  
[line 532] 540  
[line 533] 541  
[line 534] 542  
[line 535] 543  
[line 536] 544  
[line 537] 545  
[line 538] 546  
[line 539] 547  
[line 540] 548  
[line 541] 549  
[line 542] 550  
[line 543] 551  
[line 544] 552  
[line 545] 553  
[line 546] 554  
[line 547] 555  
[line 548] 556  
[line 549] 557  
[line 550] 558  
[line 551] 559  
[line 552] 560  
[line 553] 561  
[line 554] 562  
[line 555] 563  
[line 556] 564  
[line 557] 565  
[line 558] 566  
[line 559] 567  
[line 560] 568  
[line 561] 569  
[line 562] 570  
[line 563] 571  
[line 564] 572  
[line 565] 573  
[line 566] 574  
[line 567] 575  
[line 568] 576  
[line 569] 577  
[line 570] 578  
[line 571] 579  
[line 572] 580  
[line 573] 581  
[line 574] 582  
[line 575] 583  
[line 576] 584  
[line 577] 585  
[line 578] 586  
[line 579] 587  
[line 580] 588  
[line 581] 589  
[line 582] 590  
[line 583] 591  
[line 584] 592  
[line 585] 593  
[line 586] 594  
[line 587] 595  
[line 588] 596  
[line 589] 597  
[line 590] 598  
[line 591] 599  
[line 592] 600  
[line 593] 601  
[line 594] 602  
[line 595] 603  
[line 596] 604  
[line 597] 605  
[line 598] 606  
[line 599] 607  
[line 600] 608  
[line 591] 609  
[line 592] 610  
[line 593] 611  
[line 594] 612  
[line 595] 613  
[line 596] 614  
[line 597] 615  
[line 598] 616  
[line 599] 617  
[line 600] 618  
[line 601] 619  
[line 602] 620  
[line 603] 621  
[line 604] 622  
[line 605] 623  
[line 606] 624  
[line 607] 625  
[line 608] 626  
[line 609] 627  
[line 610] 628  
[line 611] 629  
[line 612] 630  
[line 613] 631  
[line 614] 632  
[line 615] 633  
[line 616] 634  
[line 617] 635  
[line 618] 636  
[line 619] 637  
[line 620] 638  
[line 621] 639  
[line 622] 640  
[line 623] 641  
[line 624] 642  
[line 625] 643  
[line 626] 644  
[line 627] 645  
[line 628] 646  
[line 629] 647  
[line 630] 648  
[line 631] 649  
[line 632] 650  
[line 633] 651  
[line 634] 652  
[line 635] 653  
[line 636] 654  
[line 637] 655  
[line 638] 656  
[line 639] 657  
[line 640] 658  
[line 641] 659  
[line 642] 660  
[line 643] 661  
[line 644] 662  
[line 645] 663  
[line 646] 664  
[line 647] 665  
[line 648] 666  
[line 649] 667  
[line 650] 668  
[line 651] 669  
[line 652] 670  
[line 653] 671  
[line 654] 672  
[line 655] 673  
[line 656] 674  
[line 657] 675  
[line 658] 676  
[line 659] 677  
[line 660] 678  
[line 661] 679  
[line 662] 680  
[line 663] 681  
[line 664] 682  
[line 665] 683  
[line 666] 684  
[line 667] 685  
[line 668] 686  
[line 669] 687  
[line 670] 688  
[line 671] 689  
[line 672] 690  
[line 673] 691  
[line 674] 692  
[line 675] 693  
[line 676] 694  
[line 677] 695  
[line 678] 696  
[line 679] 697  
[line 680] 698  
[line 681] 699  
[line 682] 700  
[line 683] 701  
[line 684] 702  
[line 685] 703  
[line 686] 704  
[line 687] 705  
[line 688] 706  
[line 689] 707  
[line 690] 708  
[line 691] 709  
[line 692] 710  
[line 693] 711  
[line 694] 712  
[line 695] 713  
[line 696] 714  
[line 697] 715  
[line 698] 716  
[line 699] 717  
[line 700] 718  
[line 690] 719  
[line 691] 720  
[line 692] 721  
[line 693] 722  
[line 694] 723  
[line 695] 724  
[line 696] 725  
[line 697] 726  
[line 698] 727  
[line 699] 728  
[line 700] 729  
[line 690] 730  
[line 691] 731  
[line 692] 732  
[line 693] 733  
[line 694] 734  
[line 695] 735  
[line 696] 736  
[line 697] 737  
[line 698] 738  
[line 699] 739  
[line 700] 740  
[line 690] 741  
[line 691] 742  
[line 692] 743  
[line 693] 744  
[line 694] 745  
[line 695] 746  
[line 696] 747  
[line 697] 748  
[line 698] 749  
[line 699] 750  
[line 700] 751  
[line 690] 752  
[line 691] 753  
[line 692] 754  
[line 693] 755  
[line 694] 756  
[line 695] 757  
[line 696] 758  
[line 697] 759  
[line 698] 760  
[line 699] 761  
[line 700] 762  
[line 690] 763  
[line 691] 764  
[line 692] 765  
[line 693] 766  
[line 694] 767  
[line 695] 768  
[line 696] 769  
[line 697] 770  
[line 698] 771  
[line 699] 772  
[line 700] 773  
[line 690] 774  
[line 691] 775  
[line 692] 776  
[line 693] 777  
[line 694] 778  
[line 695] 779  
[line 696] 780  
[line 697] 781  
[line 698] 782  
[line 699] 783  
[line 700] 784  
[line 690] 785  
[line 691] 786  
[line 692] 787  
[line 693] 788  
[line 694] 789  
[line 695] 790  
[line 696] 791  
[line 697] 792  
[line 698] 793  
[line 699] 794  
[line 700] 795  
[line 690] 796  
[line 691] 797  
[line 692] 798  
[line 693] 799  
[line 694] 800  
[line 695] 801  
[line 696] 802  
[line 697] 803  
[line 698] 804  
[line 699] 805  
[line 700] 806  
[line 690] 807  
[line 691] 808  
[line 692] 809  
[line 693] 810  
[line 694] 811  
[line 695] 812  
[line 696] 813  
[line 697] 814  
[line 698] 815  
[line 699] 816  
[line 700] 817  
[line 690] 818  
[line 691] 819  
[line 692] 820  
[line 693] 821  
[line 694] 822  
[line 695] 823  
[line 696] 824  
[line 697] 825  
[line 698] 826  
[line 699] 827  
[line 700] 828  
[line 690] 829  
[line 691] 830  
[line 692] 831  
[line 693] 832  
[line 694] 833  
[line 695] 834  
[line 696] 835  
[line 697] 836  
[line 698] 837  
[line 699] 838  
[line 700] 839  
[line 690] 840  
[line 691] 841  
[line 692] 842  
[line 693] 843  
[line 694] 844  
[line 695] 845  
[line
```

Constrained Templates (Concepts)

- NEW! in C++20: Concepts.
- Define requirements of types checkable at compile-time to be used with templates.
- Constrain the types usable with a template.
 - Better error messages!
 - Manage function and class template overload sets *much* easier.
 - This is called "subsumption"

```
#include <vector>

// A custom concept that check if a type satisfies the
// requirements of a forward container (same as the STL)
template <typename Container>
concept forward_container = requires(
    Container c, const Container cc
) {
    // Does a variable "c" of type Container have a
    // begin() method that returns the correct type?
    // Repeat for the other range methods.
    {c.begin()} -> std::same_as<typename Container::iterator>;
    {c.end()} -> std::same_as<typename Container::iterator>;
    {c.cbegin()} -> std::same_as<typename Container::const_iterator>;
    {c.cend()} -> std::same_as<typename Container::const_iterator>;

    // Does a variable cc with type const Container satisfy
    // .begin() and .end() being const-correct?
    {cc.begin()} -> std::same_as<typename Container::const_iterator>;
    {cc.end()} -> std::same_as<typename Container::const_iterator>;
};

// Result of a concept check is a boolean :::
// Can statically assert the result.
static_assert(forward_container<std::vector<int>>);
```

What is a Concept?

- A concept is a compile-time evaluable boolean expression about properties of a type.
- It is always a namespace-scope template.
- Allows us to write code that encapsulates "named requirements"
- What does it mean for a type to be Sortable?
 - It must be a sequential container.
 - It must have random access iterators
 - Each element is comparable with operator<.
- With concepts, this is writable and checkable by the compiler!

```
#include <type_traits>

namespace nonstd {
    template <typename T>
        // parentheses around the sizeof clause is mandatory
    concept bit32 = std::is_integral_v<T> && (sizeof(T) <= 4);

    template <typename T>
    concept random_access_iterator = /* requirements */;

    template <typename T>
    concept sequential_container = /* requirements */;

    template <typename T>
    concept lt_comparable = /* requirements */;

    template <typename T>
    concept sortable =
        sequential_container<T> &&
        random_access_iterator<typename T::iterator> &&
        lt_comparable<typename T::iterator::value_type>;
}
```

Custom Concepts

- It is possible for a concept to be made up only of type traits.
- Custom requirements written within a requires expression.
 - requires let's you define 0 or more variables with arbitrary types related to the template parameters of the concept.
 - Inside of the requires expression, all requirements must either be:
 - a simple requirement (some syntax that is well-formed); or
 - a type requirement (a nested typename exists); or
 - a nested requirement (e.g. requires other_concept<T>); or
 - a **compound** requirement.
 - At the end, the logical conjunction of all statements are taken.
 - I.e., all the statements are && together.
- A compound requirement is:
 - An expression wrapped in {} that should be well-formed.
 - Can optionally check the return value type as well.

Custom Concept Examples

```
#include <concepts>

// Custom concept that models an arithmetic type.
template <typename T>
concept arithmetic = requires(T t1, T t2) {
    // a nested requirement
    requires std::regular<T>;

    // two simple requirements
    sizeof(T) <= 8;
    t1 % t2;

    // four compound requirements
    {t1 + t2} -> std::same_as<T>;
    {t1 - t2} -> std::same_as<T>;
    {t1 * t2} -> std::same_as<T>;
    {t1 / t2} -> std::same_as<T>;
};
```

```
#include <concepts>

// Custom concept that models an arithmetic type.
template <typename T>
concept forward_iter = requires(T it) {
    // a nested requirement
    requires std::regular<T>;

    // a simple requirement
    // ("it" is explicitly destructible)
    it.~T();

    // four type requirements
    typename T::iterator_category;
    typename T::value_type;
    typename T::reference;
    typename T::pointer;
    typename T::difference_type;

    // four compound requirements
    {++it} -> std::same_as<T&>;
    {it++} -> std::same_as<T>;
    {*it} -> std::same_as<typename T::reference>;

    // operator-> is callable
    // (not checking return type)
    {it.operator->()};
```

What is a Good Concept?

- Concepts can check any piece of syntax.
 - May be tempting to define every operation as its own concept and to link them together in a long logical conjunction.
 - **Don't do this.**
- As a general guideline, if a concept's name captures a microscopic idea and ends in "-able" (e.g., addable), it is probably not a good concept in and of itself.
- A concept should capture a whole, practical idea, e.g.:
 - Whether a type is arithmetic, integral, or floating point.
 - Whether a type models the STL's iterator convention.
 - Whether a type can be sorted.
 - Whether a type is contiguously stored.

Constraining Templates

- With concepts, there are two ways to constrain templates.
- Constrain the template parameter directly in the template parameter list.
 - Short and easy.
 - Multi-parameter concepts: the template parameter is automatically passed as the first argument.
- Assert properties about the template parameter after the parameter list.
 - More flexible than constraining in the parameter list.
 - All template varieties (function, class, etc.) can be constrained.

```
#include <concepts>

// Guaranteed to only work with
// integers, longs, etc.
template <std::integral T>
T add(const T &t1, const T &t2) {
    return t1 + t2;
}
```

```
// THESE TWO ARE EQUIVALENT

template <typename T>
T add (const T &t1, const T & t2)
requires std::integral<T> {
    return t1 + t2;
}
```

```
#include <concepts>

// Guaranteed to only work with
// types convertible to long
template <std::convertible_to<long> T>
long add(const T &t1, const T &t2) {
    return static_cast<long>(t1) +
static_cast<long>(t2);
}
```

```
// THESE TWO ARE EQUIVALENT

template <typename T>
long add(const T &t1, const T &t2)
requires std::convertible_to<T, long> {
    return static_cast<long>(t1) +
static_cast<long>(t2);
}
```

requires Clause

- When defining a template, it can be constrained with a pre-existing concept.
- But it can also be constrained with an inline requires clause.
- If the constraint is complex, can use requires requires.
 - Should be used sparingly.
 - Better to create a custom concept.

```
#include <concepts>

// Integral is just a name!
// Let's implement what it means
// to be integral with requires requires.
template <typename Integral>
Integral add(Integral i1, Integral i2)
requires requires {
    requires std::regular<Integral>;
    std::is_integral_v<Integral>;
} {
    return i1 + i2;
}

// This could have been avoided if
// we had made a custom concept OR
// we had used std::integral instead...
```

Constrained auto

- auto is actually a concept!
 - It is the weakest concept: it has no requirements.
- It is possible to constrain auto using concepts.
 - Can be done on function template parameter types.
 - Can be done on regular variables.

```
#include <concepts>
#include <iostream>

std::floating_point auto add(
    std::floating_point auto f1,
    std::floating_point auto f2
) {
    return f1 + f2;
}

int main() {
    // OK: two floats and both are floating point.
    std::floating_point auto f = add(42.0f, 6771.0f);

    // OK: two doubles and both are floating point.
    std::floating_point auto d = add(3.14, 2.71);

    // OK: float & double both floating point!
    // float is convertible to double
    std::floating_point auto mixed = add(3.14f, 2.71);

    std::cout << std::boolalpha;
    std::cout << std::is_same_v<decltype(f), float> << '\n'
    << std::is_same_v<decltype(d), double> << '\n'
    << std::is_same_v<decltype(mixed), double> << '\n';

    // won't compile -- ints are not floating point!!!
    // std::floating_point auto error = add(42, 3.0);
} // Output: true true true
```

Overloading Based on Concepts

- It is possible to manage a template's overload set with concepts.
 - Function templates: candidate function set can be controlled
 - Class templates & variable templates: specialisation set can be controlled.
- To accomplish this:
 - The compiler normalises the requirements of the relevant templates into **conjunctive normal form**.
 - It then undergoes a process of requirement *subsumption*:
 - More constrained templates **subsume** lesser-constrained ones.
 - At the end, if more than one template remains, the usage is ambiguous.

```
#include <concepts>
#include <type_traits>

// Pre-C++20: overloading with enable_if
template <typename T> /* 1 */
std::enable_if_t<std::is_integral_v<T>, T>
secret_algorithm(T i) { return i * 2; }

template <typename T> /* 2 */
std::enable_if_t<std::is_floating_point_v<T>, T>
secret_algorithm(T fp) { return fp * T{3.14}; }

// Post-C++20: overloading with concepts
template <std::integral Int> /* 3 */
Int secret_algorithm(Int i) { return i * 2; }

template <std::floating_point Fp> /* 4 */
Fp secret_algorithm(Fp fp) { return fp * Fp{3.14L}; }

int main() {
    // Calls (3)
    int secret_int = secret_algorithm(6771);

    // Calls (4)
    float secret_float = secret_algorithm(3.14f);
}
```

Standard Concepts

- Standard Concepts can be found in a few places:
 - General purpose concepts (`std::same_as`, etc.): `<concepts>`
 - Iterator concepts (`std::bidirectional_iterator`, etc.): `<iterator>`
 - All of the `std::ranges` library's concepts: `<ranges>`
- As of C++20, there are only a few standard concepts.
 - Many more expected to be added in C++23 and beyond.
- It is encouraged to leverage the Standard Library's concepts when defining your own.
 - Code reuse > reinventing the wheel.

Feedback (stop recording)

