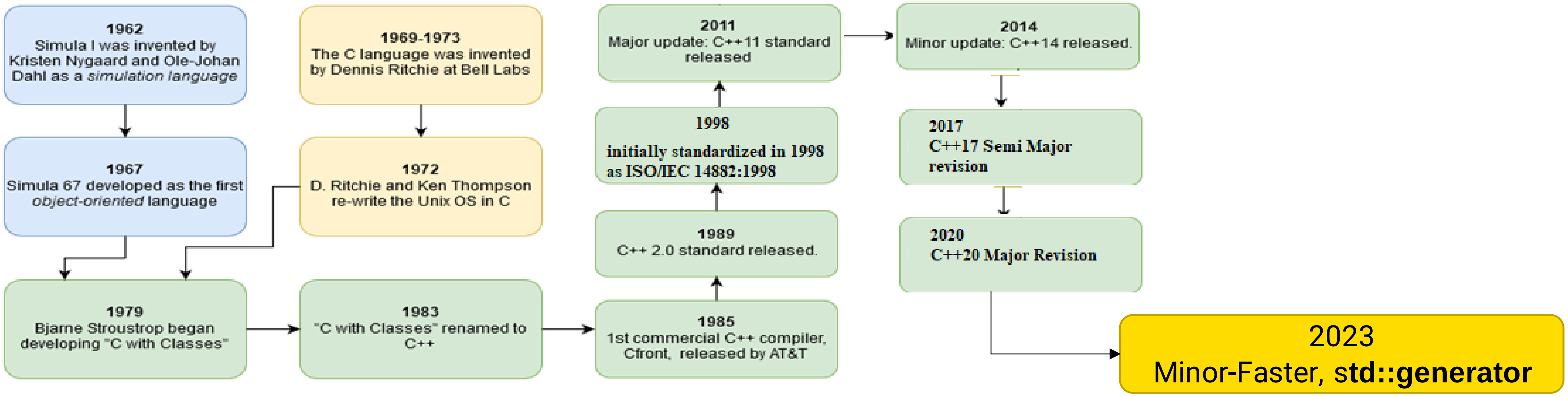
COMP6771
Advanced C++ Programming

1.2 C++ Fundamentals

C++ Standards

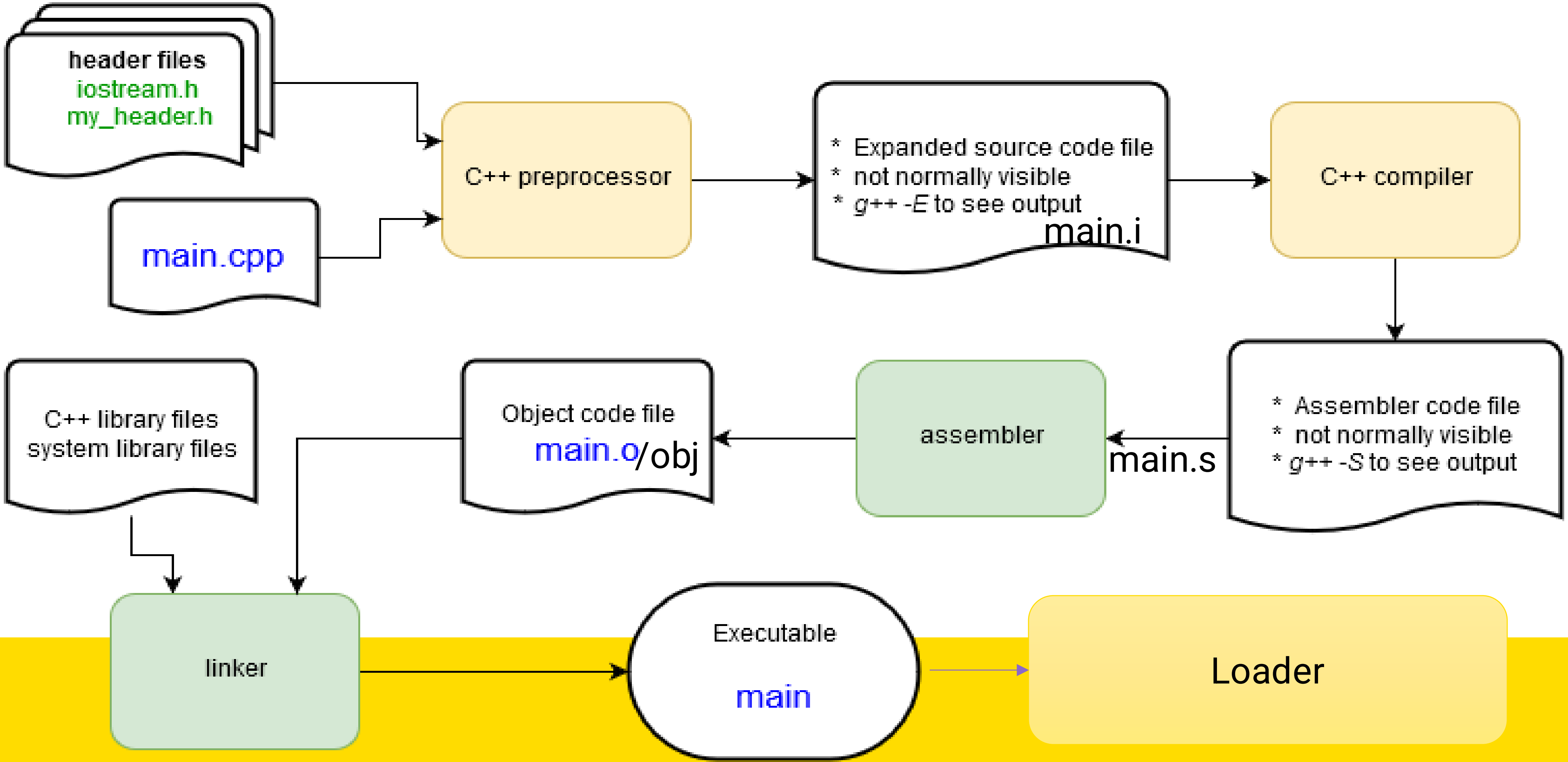
- C++ is an International Standards Organisation (ISO) language.
- Original standard was released in 1998, known as C++98.
- Since 2011, a new revision of the standard has been released every 3 years.
- 

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

Behind the Scene

```cpp
#include <iostream>

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



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.

loads a *header* file containing function and class definitions

```
#include <iostream>

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

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()

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

- 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 **struct**s, **class**es, and **union**s.
- 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

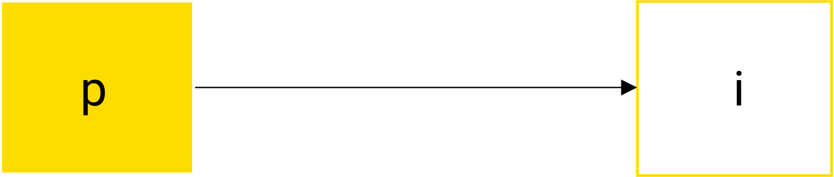
```cpp
// west const
const int ci = 42;
// east const
int const ic = 6771;
// the below will not compile.
ci++;

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!
- **A**lmost **A**lways **A**uto (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.

```cpp
#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.

```cpp
#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::end;
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)