Short Circuit

C++

A Tour of C++
by Bjarne Stroustrup
Second Edition
Up until Section 13.3

Summary by Emiel Bos

1 Basics

Source files have to be processed by a compiler, producing object files, which are combined by a linker yielding an executable program for a specific hardware/system combination.
When we talk about portability of C++ programs, we usually mean portability of source code.
The C++ standard library is implemented in C++ itself (with very minor uses of machine code for things such as thread context switching).
C++ is a statically typed language; the type of every entity (e.g., object, value, name, and expression) must be known to the compiler at its point of use.

1.1 Programs

1.1.1 Minimal program

`int main(){ }` is the minimal program. The `int` return value indicates success (`0`) or failure (nonzero) to the system. Windows-based environments rarely make use of this value, but Linux/Unix-based environments do.

One way to minimize surprises from uncaught exceptions is to use a `main()` with a `try`-block as its body.

1.1.2 Statement groups

`{ }` groups statements in C++.

1.1.3 Namespaces

A declarative region that provides a scope to the identifiers inside it and a method for preventing name conflicts in large projects. Works with block like `namespace name { }`. All blocks of the same name are in the same scope. `using namespace std;` makes names from `std` visible without `std::`.

1.1.4 Resource

Anything that has to be acquired and (explicitly or implicitly) released after use. Examples are memory, locks, sockets, file handles, and thread handles.

1.1.5 Static allocation

The memory for your variables is allocated when the program starts. The size is fixed when the program is created. It applies to global variables, file scope variables, and `static` variables defined inside functions.

1.1.6 Automatic allocation

The memory for (non-`static`) variables inside functions is allocated automatically on the *stack*. In practice, you don't have to reserve extra memory using them, but you have limited control over the lifetime of this memory.

1.1.7 Dynamic allocation

The memory for variables initialized with `new` is allocated on the *heap/free store*. In practice, you now control the exact size and the lifetime of these memory locations, which comes with the responsibility of deallocating the memory, or you'll get memory leaks (which may lead to crashes when memory is depleted).

1.2 Types, variables and operators

1.2.1 Type

Defines a set of possible values and a set of operations (for an object).

1.2.2 Object

Some memory that holds a value of some type

1.2.3 Value

A set of bits interpreted according to a type.

1.2.4 Variable

A named object.

1.2.5 `sizeof()`

The size of a type is implementation-defined and can be obtained by the `sizeof(type)` operator as the number of bytes. Typical sizes are listed.

1.2.6 Fundamental types and literals

- `bool` - 1 byte / 8 bits; boolean, possible values are `true` and `false`.
- `char` - 1 byte / 8 bits; character, for example, `'a'`, `'z'`, and `'9'`.
- `int` - 4 bytes / 32 bits; integer, for example, `-273`, `42`, and `1066`.
- `float` - 4 byte / 32 bits; floating-point number, for example, `5.f`, and `1/3.f`.
- `double` - 8 byte / 64 bits (at least as large as `float`); double-precision floating-point number, for example, `-273.15`, `3.14`, and `6.626e-34`. The default for literals with a decimal point.
- `long double` - (at least as large as `double`).
- `unsigned`; non-negative integer, for example, `0`, `1`, and `999` (use for bitwise logical operations).
- `const char[]`; C-style string, for example, `"Surprise!"`.
- `std::string`; standard library `std::string`, for example, `"Surprise!"s`.
- `std::string`; standard library `std::string_view`, for example, `"Surprise!"sv`.

You need to import the `std::literals::string_literals` namespace to use the `s` suffix, and `std::literals::string_view_literals` for the `sv` suffix.

1.2.7 Integer literals

- default; decimal (base 10), (e.g.`42`)
- `0b` prefix; binary (base 2), (e.g. `0b10101010`)
- `0x` prefix; hexadecimal (base 16), (e.g. `0xBAD1234`)
- `0` prefix; octal (base 0), (e.g. `0334`)
- `u` suffix; unsigned int, (e.g. `0xFF00u`)
- `l` suffix; long int, (e.g. `0xFF00u`)
- `ul` suffix; unsigned long int, (e.g. `0xFF00u`)
- `ll` suffix; long long int, (e.g. `0xFF00u`)

- `ull` suffix; unsigned long long int, (e.g. `0xFF00u`)
- `s` suffix; `second`, (e.g. `123s`)
- `i` suffix; `imaginary`, (e.g. `13 + 42i` is `std::complex{13, 42i}`)

1.2.8 `char`

Single-byte built-in type representing a single ASCII character. Implicitly converts to `int` if needed, in which case its value is its ASCII value.

1.2.9 Integer single-quote operator

To make long literals more readable for humans, we can use a single quote (') as a digit separator: 3.14159'26535'89793' 23846'26433'83279'50288 or 0x3.243F'6A88'85A3'08D3.

1.2.10 Integer-boolean relation

If an integer is `0`, it converts to `false`, else it is `true`. Conversely, converting `0` integers to booleans yields `false`, converting any other value yields `true`.

1.2.11 Arithmetic operators

- `x+y`; plus
- `+x`; unary plus
- `x-y`; minus
- `-x`; unary minus
- `x*y`; multiply
- `x/y`; divide
- `x\%y`; modulo/remainder for integers
- `x<<i`; bitshift i bits to the left
- `x>>i`; bitshift i bits to the right
- `x+=y`; x = x+y
- `x-=y`; x = x-y
- `++x`; increment: x = x+1, returns incremented x as lvalue
- `x++`; increment: x = x+1, returns old x
- `--x`; decrement: x = x-1, returns decremented x as lvalue
- `x--`; decrement: x = x-1, returns old x
- `x*=y`; scaling: x = x*y
- `x/=y`; scaling: x = x/y
- `x\%=y`; x = x%y

1.2.12 Comparison operators

- `x==y`; equal
- `x!=y`; not equal
- `x<y`; less than
- `x>y`; greater than
- `x<=y`; less than or equal
- `x>=y`; greater than or equal

1.2.13 Logical operators

- `x&y`; bitwise and
- `x|y`; bitwise or
- `x^y`; bitwise exclusive or
- `~x`; bitwise complement
- `x&&y`; logical and
- `x||y`; logical or
- `!x`; logical not (negation)

1.2.14 Usual arithmetic conversions

In assignments and in arithmetic operations, C++ performs all meaningful conversions between the basic types so that they can be mixed freely. The conversions used in expressions are called the *usual arithmetic conversions* and aim to ensure that expressions are computed at the highest precision of its operands.

1.2.15 Assignment

An assignment of a built-in type is a simple machine copy operation, with the operands remaining independent, unlike languages like Python, C# and Java. If you want different objects to refer to the same (shared) value, use pointers or references.

1.2.16 Initialization

In general, for an assignment to work correctly, the assigned-to object must have a value, while initialization is meant to make an uninitialized piece of memory into a valid object. For almost all types, the effect of reading from or writing to an uninitialized variable is undefined.

```
double d1 = 2.3; //initialize d1 to 2.3
double d2 {2.3}; //initialize d2 to 2.3
double d3 = {2.3}; //initialize d3 to 2.3 (the = is optional with { ... })
complex<double> z = 1; //a complex number with double-precision floating-point scalars
complex<double> z2 {d1,d2};
complex<double> z3 = {d1,d2}; //the = is optional with { ... }
vector<int> v {1,2,3,4,5,6}; //a vector of ints
```

Conversions that lose information (*narrowing conversions*) are allowed and implicitly applied when you use `=`, but not when you use `{}`. The problems caused by implicit narrowing conversions are a price paid for C compatibility.
Try to avoid leaving variables unitialized. User-defined types can be implicitly initialized.
The basic semantics of argument passing and function value return are that of initialization.

1.2.17 `auto`

A "type" qualifier that deduces the actual type. With `auto`, we tend to use the = because there is no potentially troublesome type conversion involved. Can avoid redundancy and writing long type names, but may be less readable. Can also be used for return types of functions (though this does not offer a stable interface; a change to the implementation of the function – or lambda – can change the type).

1.2.18 lvalue

Roughly means "something that can appear on the left-hand side of an assignment". An lvalue has a memory address. It's essentially a container.

1.2.19 rvalue

Roughly means "something that can only appear on the right-hand side of an assignment", or, "a value that you can't assign to". Literals and function calls that return local variables by value are rvalues. An rvalue does not have a memory address (except for some temporary register), and is short-lived and temporary. It's essentially something to put in a container. Normally only used to declare a parameter of a function.

1.2.20 rvalue reference (`&&`)

A reference that can only be assigned an rvalue, e.g. a `5` or a `'v'` can be assigned to an rvalue reference variable or passed to an rvalue reference parameter, but another variable cannot. A reference to something that nobody else can assign to, so we can safely "steal" its value. Local variables in functions [that are meant to be returned?] are an example. Used in move constructors and move assignment.

1.2.21 `using` alias

Define an alias for a type with `using`:

```
using size_t = unsigned int; // The actual type size_t is implementation dependent and could be
    unsigned long for example.
```

Also used for read-only access to template arguments.

1.2.22 `typdef`

Almost the same as a `using` alias, i.e. they have the same semantics, with the difference that an alias declaration is compatible with templates, whereas the C style `typdef` is not. However, `typdef` declarations can be used as initialization statements, whereas alias declarations cannot. `using` aliases seem to be the recommended usage.

1.3 Scope and lifetime

An object must be constructed (initialized) before it is used and will be destroyed at the end of its scope.

1.3.1 Local scope

Names declared in a function or lambda that aren't initialized with `new` and function argument names are *local names*. Extends from its point of declaration to the end of the block (delimited by a `{}` pair) in which its declaration occurs. Global variables are automatically allocated.

1.3.2 Class scope

Names declared in a class/struct outside any function, lambda or `enum class` are *(class) member names*. Extends from the opening `{` of its enclosing declaration to the end of that declaration. The point of destruction of members is determined by the point of destruction of the object of which it is a member.

1.3.3 Namespace scope

Names defined in a namespace outside any function, lambda, class, or `enum class` are *namespace member names*. Extends from the point of declaration to the end of its namespace. The point of destruction of namespace objects is the end of the program.

1.3.4 Global scope

Names not declared inside any other construct are *global names* and are in the *global namespace*. Extends from the point of declaration to the end of its namespace or the end of its program. Global variables are statically allocated.

1.3.5 Unnamed objects

Temporaries and objects created using `new`, e.g. `auto p = new Record{"Hume"};` (`p` points to an unnamed `Record`). An object created by `new` "lives" until destroyed by `delete`.

1.4 Constants

1.4.1 `const`

Keyword used primarily to specify interfaces that guarantee that a variable will not be changed. `const` values can be calculated at run time.

1.4.2 `constexpr`

Keyword used primarily to specify constants, to allow placement of data in read-only memory (where it is unlikely to be corrupted), and for performance. `constexpr` values must be calculated by the compiler.

```
vector<double> v {1.2, 3.4, 4.5}; // v is not a constant
const double s1 = sum(v); // OK: sum(v) is evaluated at run time
constexpr double s2 = sum(v); // error : sum(v) is not a constant expression
```

Constant expressions are required by array bounds, case labels, template value arguments [hyperref TBD], and constants declared using `constexpr`. Compile-time evaluation is important for performance.

1.4.3 `constexpr` functions

For a function to be usable in a *constant expression* (i.e. evaluated by the compiler) it must be defined `constexpr`:

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

and must be called with constant expressions. Of course, you can call it with non-`constexpr` arguments, but then the result won't be a constant expression either. `constexpr` functions cannot modify non-local variables.

1.4.4 `const` member functions

A `const` specifier on class member function indicate that it does not modify its object. A `const` member function can be invoked for both `const` and non-`const` objects (obviously), but a non-`const` member function can only be invoked for non-`const` objects.

```
class SomeClass {
   public:
      SomeClass(int num) : number = num { }
      int getNumber() const { return number; } // Doesn't change its SomeClass object
      int setNumber(int num) const { number = num; } // Can't be const
   private:
      int number;
};
```

1.5 Arrays, pointers and references

1.5.1 Postfix unary `*`

Pointer to postfixed type.

1.5.2 Postfix unary `&`

Reference to postfixed type.

1.5.3 Prefix unary `*` (dereference operator)

"Contents of" expression/pointer. Dereferences whatever `*` prefixes.

1.5.4 Prefix unary `&` (address-of operator)

"Address of" expression/variable.

1.5.5 Pointer

`char* p = &var;` declares `p` initialized to point to `var`.
`char x = *p;` declares `x` initialized as the `char` that `p` points to.
A pointer is a machine address.

1.5.6 `std::unique_ptr`

A *smart pointer* (pointers designed to prevent memory leaks) that automatically calls `delete` in its pointed-to object once it goes out of scope.

```
unique_ptr<SomeType> uptr; // An alternative to SomeType* uptr;
unique_ptr<SomeType> uptr2(new SomeType); // Allocate a new SomeType and give its pointer to uptr2

auto uptr3 = std::make_unique<SomeType>(); // Less verbose and less error-prone way, which avoids
    the heap allocation and pointer passing. You directly call SomeType's constructor.
```

Can of course be used to pass free-store allocated objects in and out of functions. `unique_ptr`'s have no time or space overhead. Since a `std::unique_ptr` is the sole owner of the object, it can't be copied:

```
auto p = make_unique<int>(2);
auto q = p; // Error
```

1.5.7 `std::shared_ptr`

A smart pointer, similar to `unique_ptr`s except that `shared_ptr`s are copied rather than moved. Multiple `shared_ptr`s share ownership of an object, and this object is automatically destructed once the last `shared_ptr` is destructed.

```
shared_ptr<SomeType> sptr(new SomeType());
function1(sptr); // Copy by value, so function1 has a different shared_ptr pointing to the same
    SomeType
function2(sptr); // Idem. SomeType will be destructed by the last function to (explicitly or
    implicitly) destroy a copy of sptr

auto sptr2 = std::make_shared<SomeType>(); // Less verbose and less error-prone way, which avoids
    the heap allocation and pointer passing. You directly call SomeType's constructor. Also notably
    more efficient because it does not need a separate allocation for the use count
```

`shared_ptr`s have minimal overhead, but do make the lifetime of the shared object hard to predict, so use them sparingly.

1.5.8 (C) Array

A contiguously allocated sequence of elements of the same type. The most fundamental collection of data. Basically what the hardware offers.
`char arr[6];` declares array `arr` of six `char`s with elements `arr[0]` through `arr[5]`. `int arr[5] = {1,2,3,4,5};` intitializes the array with the given values.
With the exception of `static` arrays and arrays in file scope (outside of all functions), it will be allocated *on the stack*, which only has a limited size.
`char* arr = new char[6];` is similar, but allocates the array on the heap.
An array variable is actually a pointer to its first element.
The size of an array must be a `constexpr`.
Using C arrays is nearly always a bad idea; likely you'll want to use a `std::vector<>` or `std::array<>`.

1.5.9 Pointer into array

An array variable name is a (or degenerates to [?]) constant pointer to the first element of the array. Therefore, it is legal to use array names as constant pointers, and vice versa.
We can advance a (non-`const`) pointer into an array to point to the next element of an array using `++`.

1.5.10 `nullptr`

For when we don't have an object to point to or if we need to represent the notion of "no object available" (e.g. for an end of a list). Dereferencing is invalid. We can check using `if (p == nullptr)`.

1.5.11 Pointer-boolean relation

If a pointer is the `nullptr`, it converts to `false`, else it is `true`.

1.5.12 Reference

`char& r = var;` declares `r` initialized as an (lvalue) reference to `var`.
Similar to a pointer, but no need to use a prefix * to access the value referred to.
A reference cannot be made to refer to a different object after its initialization.
There is no "null reference"; a reference must always refer to a valid object.
A reference is a machine address, like a pointer. The standard places no requirements on how compilers implement references (they probably use pointers under the hood).

1.6 Functions

1.6.1 Declaration

A function cannot be called unless it has been previously declared (and elsewhere defined): `double pow(double, int);`
Declarations constitute an interface and specify all that's needed to use a function or a type. A function declaration may contain argument names, which are ignored (unless it's a definition), but can make it more readable.

1.6.2 Function types

The type of a function consists of its return type and the sequence of its argument types:

```
double get(const vector<double>& vec, int index); //type: double(const vector<double>&, int)
```

For a member function, the name of its class is also part of the function type:

```
char& String::operator[](int index); //type: char& String::(int)
```

1.6.3 `void`

A "return type" that indicates that a function does not return a value.

1.6.4 Function overloading

Essential parts of generic programming. If two functions are defined with the same name but different argument types/signatures, the compiler will choose the most appropriate function to invoke for each call. If two alternative functions could be called, but neither is better than the other, the call is deemed ambiguous and the compiler gives an error:

```
void print(int,double);
void print(double ,int);

void user() {
   print(0,0); // error: ambiguous
}
```

If the functions involved are templates with concepts, the compiler will select the template with the strongest requirements met by the arguments if everything is equal (or it gives an ambiguity error).

1.6.5 `operator` overloading

Operators are basically functions with symbols as names. User-defined operators (*overloaded operators*) look like this:

```
class SomeClass {
   public:
      SomeClass(int num) : number = num { }
      SomeClass& operator+=(SomeClass r) {
         number += r.number;
         return *this;
      }
   private:
      int number;
};
```

Even the function call operator `operator()` or indexing operator `operator[]` can be overloaded.

1.6.6 Prefix and suffix overloading

`operator""` indicates that we are defining a literal operator, which converts a literal of its argument type – followed by a subscript – into its return type.

```
constexpr complex<double> operator""i(long double arg) {
   return {0, arg};
}
```

Now we can write:

```
omplex<double> z = 2.7 + 6.25i;
```

If an operator does not require direct access to the representation of `SomeClass`, it can be defined separately from the class definition:

```
SomeClass operator+(SomeClass l, SomeClass r) { return l+=r; }
SomeClass operator-(SomeClass sc) { return {-sc.number}; } // Unary minus
```

The syntax is fixed by the language, so you can't define a unary `/`. Also, it is not possible to change the meaning of an operator for built-in types, so you can't redefine `+` to subtract `int`s.

1.6.7 Pass-by-value

By default, arguments are copied and become local variables if there are no qualifiers or keywords present. Can be more performant than passing-by-reference if the variable is small enough, where "small" is machine-dependent but the size of three pointers or less is a good rule of thumb.

1.6.8 Pass-by-reference

`void sort(vector<double>& v);` sorts `v` itself (via a reference) and not a copy. This is more efficient in case the `vector` is very large.

1.6.9 Pass-by-const-reference

`double sum(const vector<double>&);`. When you don't want to modify an argument but still don't want the cost of copying. Very common.

1.6.10 Default function argument

```
void function(int value = 10); //value == 10 if no argument is supplied.
```

1.6.11 Return-by-value

The default for value return is to copy and for small objects that's ideal.

1.6.12 Return-by-reference

In case we want to grant a caller access to something that is not local to the function, i.e. something outside of the function. On the other hand, a local variable disappears when the function returns, so we should not return a pointer or reference to it, but most compilers will catch this.

1.6.13 Return-by-moving

In case you want to efficiently pass large amounts of data out of a function, create a move constructor for the type of the variable you want to `return` and `return` the value as usual. This is as efficient as returning a pointer.

1.6.14 Structured binding

For returning multiple values:

```
struct Entry {
   string name;
   int value;
};

createEntry(string name, int value) return {name, value};
```

Similarly, we can "unpack" a `struct` into local variables:

```
auto [n,v] = entry; // Declares two local variables n and v with their types deduced
```

We can also use this *structured binding* for iterating using a `for`-loop:

```
map<string,int> m;
// Fill m
for (auto [key,value] : m) // Do something with key or value. You can of course add const and/or &
```

When structured binding is used for a class with no private data, there must be the same number of names defined for the binding as there are non-static data members of the class, and each name introduced in the binding names the corresponding member.

1.6.15 `inline`

Inlining essentially replaces the function call of a simple function with its actual operation(s)/definition/function body during the generation of machine code, avoiding a function call. Precede a function declaration with the inline `inline` to explicitly request inlining. Functions defined in a class (so not member functions defined externally, outside the class body) are inlined by default.

1.7 Language constructs

1.7.1 `if`

An `if`-statement can introduce a variable and test it:

```
if (auto n = v.size(); n != 0) // n is in scope in both branches of the if-statement
```

To test a variable against `0` (or the `nullptr`), simply leave out the explicit mention of the condition:

```
if (auto n = v.size())
```

1.7.2 Compile-time `if constexpr` (C++17)

`if constexpr(is_pod<T>::value)` is evaluated at compile time, and only the selected branch is instantiated. Note that this is not a source code-manipulation mechanism and cannot be used to break the usual rules of syntax and grammar.

1.7.3 `for`

```
for (int i = 0; i < 10; ++i)
```

We can leave out the initializer if we don't need it:

```
for (;*p != 0; ++p)
```

but then you may as well use a...

1.7.4 `while`

```
while (*p)
```

1.7.5 range-`for`

For traversing a sequence. We can copy values one-by-one into `x`:

```
for (auto x : seq)
```

or get references:

```
for (auto& x : seq)
```

This uses `begin()` and `end()` implicitly and is roughly equivalent to the more explicit `for (auto p = c.begin(); p != c.end(); ++p)`.

1.7.6 `switch`

```
case 'y':
   return true;
case 'n':
   return false;
default:
   std::cout << "Whatever.\n";
   return false;
```

case-labels must be distinct and constexpr. If the value doesn't match any case-label and no default is provided, no action is taken.

1.7.7 new

Dynamically allocates memory from an area called the *free store* aka *dynamic memory* aka *heap*, and calls the constructor of the type for which it is called. new allocates and constructs a single object, while new[] does so for an array:

```
SomeClass* scp = new SomeClass;
double* numbers = new double[42];
```

Objects allocated on the free store are independent of the scope from which they are created and "live" until they are destroyed using delete.

1.7.8 delete

Deallocates memory from the *free store* aka *dynamic memory* aka *heap*. Plain delete deletes an individual object, delete[] deletes an array. Calling delete on a pointer to a user-defined type will call the type's (potentially virtual) destructor.

1.7.9 malloc()

Allocates memory, but doesn't construct an object. A remnant from C, and isn't much used in C++.

```
int *p = malloc(42); // Returns the pointer to the beginning of 42 bytes of newly allocated memory,
    or nullptr
free(p); // Don't forget to deallocate the memory.
```

1.7.10 static_cast<>()

Converts an expression to a new type:

```
static_cast<newType>(expression);
```

A static_cast does not check the value it is converting; the programmer is trusted to use it correctly. Explicit type conversions (often called casts to remind you that they are used to prop up something broken) are best avoided.

1.7.11 reinterpret_cast<>()

reinterpret_cast<newType>(expression); is a compile-time directive that doens't translate to any CPU insttruction, but simply tells the compiler to treat expression as a newType.

1.7.12 const_cast<>()

Removes const and casts to new type.

1.8 Modularity

1.8.1 Separate compilation

Types and functions used can be defined in a separate source file, which will be compiled only once and reused to avoid recompilation.

1.8.2 Header files

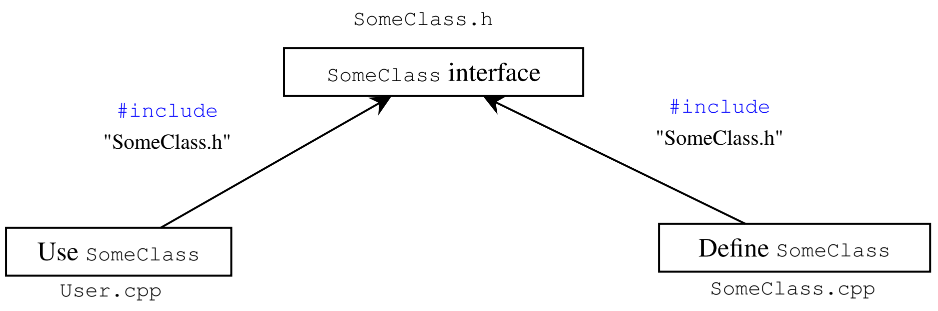
Typically, we place the declarations that specify the interface to a module in a header file ending in .h.

1.8.3 #includes

Header files can then be included in the main .cpp file with #include "Vector.h".

With *most* compilers, using `""` first checks your local directory, and if it doesn't find a match checks the system paths, while using `<>` starts the search with system headers, but this is all implementation dependent. Convention dictates to use `<>` for system headers, and `""` for your own headers.

1.8.4 Program structure



The code in `User.cpp` and `SomeClass.cpp` shares the `SomeClass` interface information presented in `SomeClass.h`, but the two files are otherwise independent and can be separately compiled.

1.8.5 `module` (C++20)

`#include`s are very old, error-prone, rather expensive, and order dependent. In `SomeClass.cpp`:

```
module; // This compilation will define a module

// Here we put (include/define) stuff that SomeClass needs

export module SomeClass; // Defining the module called "SomeClass"

export class SomeClass { // Export this class
   public:
      SomeClass(int num);
      getNumber();
   private:
      int number;
};

SomeClass::SomeClass(int num): number = num { } // Because we export SomeClass, all its member
    functions (i.e. this constructor) are automatically exported

int SomeClass::getNumber() { return number; } // Exported as well

export double nonMemberFunction(const SomeClass& c) { return c.getNumber(); } // Non-member are not
    automatically exported, so we do this explicitly
```

We can then `import SomeClass;` to use it on other files. `import`s and `#include`s can be freely used together.

The differences between headers and modules are not just syntactic:

- A module is compiled once only (rather than in each translation unit in which it is used).
- Two modules can be `import`ed in either order without changing their meaning
- If you `import` something into a module, users of your module do not implicitly gain access to (and are not bothered by) what you imported; `import` is not transitive.

1.8.6 `namespace`

Another mechanism/scope for grouping declarations and making sure their names don't clash with others.

```
namespace SomeNamespace {
   int main();
}

int SomeNamespace::main() {
    // Implementation
}

int main() { // Doesn't clash with
    return SomeNamespace::main();
}
```

Standard-library names are in namespace `std`. Names not local to a defined namespace, class, or function are in the global namespace. `using std::swap` makes `std::swap` usable as if it was declared in the scope in which it appears, i.e. no need for `std::`. `using namespace std;` makes the entire namespace available.

1.9 Exception handling

Compilers are optimized to make returning a value much cheaper than throwing the same value as an exception, but do not believe the myth that exception handling is slow; it is often faster than correct handling of complex or rare error conditions, and of repeated tests of error codes.

1.9.1 `throw`

`throw someException{"Error message"};` transfers control to a handler for exceptions of type `someException` in the function that directly or indirectly called the current function.

1.9.2 Standard library exceptions

- `std::out_of_range`
- `std::length_error`; when a non-positive number of elements is used to specify a sequence's length
- `std::bad_alloc`; when operator `new` can't find memory to allocate (memory exhaustion)

1.9.3 `try-catch`

```
try { // Exceptions here are handled by the handler defined below
    // Code that may throw
}
catch (someException& exception) {
    std::cerr << exception.what() << '\n';
}
```

`std::exception::what()` retrieves the error message.

1.9.4 Re`throw`

```
try {
    // Code that may throw
}
catch (someException& exception) {
    throw; // If this function can't complete its task, we can rethrow someException
}
```

1.9.5 `noexcept`

A function that should never throw an exception can be declared `noexcept`, e.g. `void function()noexcept;`
If it still `throw`s, `std::terminate()` is called.

1.9.6 Invariant

A statement of what is assumed to be true for a class is called a class invariant, or simply an invariant. It is the job of a constructor to establish the invariant for its class (so that the member functions can rely on it) and for the member functions to make sure that the invariant holds when they exit.

1.9.7 `assert`

`assert(p != nullptr);` asserts that a condition (valid `p`) must hold in at runtime. The program terminates when this fails in debug mode; in release mode `assert`s aren't checked.

1.9.8 `static_assert`

Like `assert` but for compile time. Can be used for anything that can be expressed as a constant expression, e.g.

```
constexpr int minIntSize = 4;
static_assert(minIntSize <= sizeof(int), "Integers are too small."); // Writes "Integers are too
    small." as a compiler error message if an int on this system does not have at least 4 bytes
```

If you don't supply a message, the compiler will write a default message (typically the source location of the `static_assert` plus a character representation of the asserted predicate) if the assertion fails. The most important uses of `static_assert` come when we make assertions about types used as parameters in generic programming.

1.9.9 Resource Acquisition Is Initialization (RAII)

The technique of acquiring resources in a constructor and releasing them in a destructor, thus making resource release guaranteed and implicit and avoiding allocations in general code and keep them buried inside the implementation of well-behaved abstractions. Let each resource have an owner in some scope and by default be released at the end of its owners scope.

In the C++ standard library, RAII is pervasive: for example, memory (`std::string`, `std::vector`, `std::map`, `std::unordered_map`, etc.), files (`std::ifstream`, `std::ofstream`, etc.), threads (`std::thread`), locks (`std::lock_guard`, `std::unique_lock`, etc.), and general objects (through `std::unique_ptr` and `std::shared_ptr`).

2 Types

Types built out of other types using C++'s abstraction mechanisms.

2.1 User-defined Types

2.1.1 Special member functions

If a class `X` has a destructor that performs a nontrivial task, such as free-store deallocation or lock release, the class is likely to need the full complement of special member functions:

```
class X {
   public:
   X(int, SomeOtherType); // Constructor
   X(); // Default constructor
   X(const X&); // Copy constructor
   X(X&&); // Move constructor
   X& operator=(const X&); // Copy assignment: clean up target and copy
   X& operator=(X&&); // Move assignment: clean up target and move
   ~X() // Destructor: clean up
   //...};
```

2.1.2 `=default` special member functions

Except for the "ordinary constructor", the special member functions will be generated by the compiler as needed. If you want to be explicit about generating default implementations, you can by using

```
X(const X&) = default; // Enforce a default copy constructor
```

If you are explicit about some defaults, other default definitions will not be generated. Explicit copy and move operations are usually a good idea when a class has a pointer member, because a pointer may point to something that the class needs to (not) `delete`, in which case the default memberwise copy would be wrong. A reader of the code would like to know. A good rule of thumb (sometimes called the *rule of zero*) is to either define all of the essential operations or none (using the default for all).

2.1.3 `=delete` member functions

Indicates that an operation is not to be generated. A base class in a class hierarchy is the classical example where we don't want to allow a memberwise copy. A `=delete` makes an attempted use of the `delete`d function a compile-time error; `=delete` can be used to suppress any function, not just essential member functions.

2.1.4 `public` member

Accessible to all code that can access/use the class of which it's a member.

2.1.5 `protected` member

Accessible only to descendents of the class of which it's a member.

2.1.6 `private` member

Accessible only to the class of which it's a member itself.

2.1.7 `struct`

By default, members are `public` in a `struct`. By convention, `struct`s serve more as a bundle of data.

```
struct Struct {
   int number;
};

Struct s;
s.number = 42; // Access through name (or reference)
Struct* sp = &s;
sp->number = 43; // Access through pointer
```

2.1.8 `class`

The only difference with a `struct` is that, by default, members are `private` in a `class` and that `class`es inherit `private`ly, whereas members default to `public` in a `struct`. `class`es are really meant for the notions of encapsulation and consolidating data and operations (on that data).

```
class SomeClass {
   public: // Constitutes the interface
      // SomeClass() { }; is implicitly generated
      SomeClass(int num) : number = num { } // Constructor
      int getNumber() { return number; }
   private: // Only accessible via the interface
      int number;
};
```

2.1.9 `this`

The name `this` is predefined in a member function and points to the object for which the member function is called.

2.1.10 Constructor

A member "function" with the same name as its class is called a constructor; a function used to construct objects of a class. A constructor that can be invoked without an argument is called a *default constructor*. A default constructor is implicitly generated if none is specified, which default initializes its members. A constructor of a derived class calls the constructor of its base class (implicitly in case there is a default constructor), meaning derived classes are constructed "bottom-up" (base first). Constructors don't [normally?] `return` anything, but you can `return` without an expression.

```
SomeClass sc; // Default initialization/construction
SomeClass sc = SomeClass(42); // Initialization
```

2.1.11 static members

Member declaration specifier that declares members (variables or functions) that are not bound to class instances. They exist even if no objects of the class have been defined; hence they are referred to by their encapsulating class name rather than a specific variable name of its type. `static` functions have no `this` pointer. Only `inline static` members may be defined in the class definition:

```
struct X
{
   static void f(); // Function declaration. Cannot be defined in-class
   static int n; // Variable declaration. Cannot be defined in-class
   inline static int in = 123; // Inline static member CAN be defined inline
};

int X::n = 69; // Variable definition
```

Outside a class definition, it has a different meaning.

2.1.12 Initializer list

Notationally shorter way of initializing a class's member variables. Not to be confused with `std::initializer_list`.

```
class Point {
private:
   int x;
   int y;
public:
   Point(int i = 0, int j = 0) : x(i), y(j) {
      // Without the initializer list, we would have to do:
      // x = i;
      // y = j;
   }
};
```

`const` data members, reference variables, member objects that do not have a default constructor, and base class members must be initialized using an initializer list.

2.1.13 Destructor

A mechanism to ensure that the memory allocated by the constructor is deallocated.

```
class SomeClass {
   public:
      ~SomeClass() { delete[] numbers; }
   private:
      int size;
      int* numbers;
};
```

Destructors are called automatically when variables of its type go out of scope or run out of lifetime. Destructors implicitly call the destructors of all member variables and base classes (meaning a derived class is destroyed "top-down", i.e. derived first) after the class body has run. Member variables are destructed in the opposite order they were constructed in, which is the order in which they appear in the class declaration. Constructor-destructor pairs are important to RAII.

2.1.14 Conversion

A constructor taking a single argument defines a conversion from its argument type, which then allows for implicit conversions.

2.1.15 explicit conversion

In case we want to avoid implicit conversions caused by a single-parameter constructor, we prepend it with `explicit` like so:

```
explicit SomeClass(int num);
```

2.1.16 Default member variables values

```
class SomeClass {
   int number = 42;
};
```

The default value is used whenever a constructor doesn't provide a value.

2.1.17 Copy

By default, objects can be copied. The default meaning of copy is memberwise copy.

```
SomeClass a = SomeClass(); // Default constructor initialization
SomeClass b{a}; // Copy initialization/construction (memberwise copy)
SomeClass c = a; // Copy assignment (memberwise copy)
```

2.1.18 Copy constructor

Used to initialize a previously uninitialized object (hence the name copy "constructor") from some other object's data. The default meaning of copy is memberwise copy, i.e. copying each member. For some sophisticated concrete types like resource handles or containers, memberwise copy is not the right semantics for copy; for abstract types it almost never is. For containers, the default copy would simply copy only the handle, leaving the two objects to refer to the same elements. We need our own copy constructor:

```
SomeContainer::SomeContainer(const SomeContainer& sc) : numbers = new int[sc.size], size = sc.size {
   for (int i = 0; i != size; ++i) // Copy elements
      numbers[i] = sc.numbers[i];
}
```

2.1.19 Copy assignment

Used to replace the data of a previously initialized object with some other object's data. A copy constructor almost always needs a corresponding copy assignment operator definition:

```
SomeContainer& SomeContainer::operator=(const SomeContainer& sc) {
   int* p = new int[sc.size];
   for (int i=0; i != sc.size; ++i)
      p[i] = sc.numbers[i];
   delete[] numbers; // Delete old elements
   numbers = p;
   size = sc.size;
   return *this;
}
```

Default copy assignment does a simple memberwise copy, which for pointer and handles can cause memory leaks (hence they are often `=delete`d for such types).

2.1.20 Move constructor

`return`ing large values is expensive if they need to be copied out of a function while there is no move constructor. Defining a move constructor allows data to be efficiently moved out of a function:

```
SomeContainer::SomeContainer(SomeContainer&& sc) : numbers = sc.numbers, size = sc.size {
   sc.numbers = nullptr;
   sc.size = 0;
}
```

A move operation is applied when an rvalue reference is used as an initializer or as the right-hand side of an assignment. This means that `return`ing an object of a type with a move constructor gets moved out of the function instead of copied out of it. After a move, a moved-from object should be in a state that allows a destructor to be run. Typically, we also allow assignment to a moved-from object.

2.1.21 `std::move()`

Where the programmer knows that a value will not be used again, but the compiler can't be expected to be smart enough to figure that out, you can use `std::move()` to get an rvalue reference to its argument from which we may move; it doesn't actually move anything, but essentially casts an lvalue to an rvalue reference (it really should have been called `rvalue_cast` or something).

```
Vector someFunction() {
   // Initialize ints x, y, z
   z = x; // Copy (because x might be used later in someFunction())
   y = std::move(x); // Move (assignment), so x should not be used anymore
   // ... better not use x here ...
   return z; // Move
}
```

The state of a moved-from object is in general unspecified, but all standard-library types leave a moved-from object in a state where it can be destroyed and assigned to, e.g. `std::vector`s and `std::string`s are empty. For many types, the default value is fine as replacement.

Like other casts, it's error-prone and best avoided. Use only if you can demonstrate significant and necessary performance improvement. It exists to serve a few essential cases, e.g. a simple three-step swap operation, where we don't want to copy three times.

2.1.22 Move assignment

A move constructor [almost?] always needs a corresponding move assignment operator definition:

```
SomeContainer& SomeContainer::operator=(const SomeContainer& sc) {
   int* p = new int[sc.size];
   for (int i=0; i != sc.size; ++i)
      p[i] = sc.numbers[i];
   delete[] numbers; // Delete old elements
   numbers = p;
   size = sc.size;
   return *this;
}
```

2.1.23 `std::initializer_list`

Not to be confused with a constructor's initializer list.

When we use a `{}`-list, such as `{1,2,3,4}`, the compiler will create an object of type `initializer_list`, which can be used to construct containers with.

```
class SomeContainer {
   public:
      SomeContainer(std::initializer_list<double> list) : numbers = new int[list.size()], sz =
          static_cast<int>(lst.size()); { copy(list.begin(), list.end(), numbers) } // Initialize
          with a list of int
   private:
      int size;
      int* numbers; // numbers points to an array of ints of size size
}

SomeContainer sc = {1,2,3};
```

Unfortunately, the standard-library uses `unsigned` integers for sizes and subscripts, so you need to use the ugly `static_cast` to explicitly convert the size of the `std::initializer_list` to an `int`, even though the chance that the number of elements in a handwritten list is larger than the largest integer is rather low.

2.1.24 `union`

A `struct` in which all members are allocated at the same address, so that the `union` occupies only as much space as its largest member.

```
union Value {
   Node* p;
   int i;
};
```

Naturally, a union can hold a value for only one member at a time, so the programmer must keep track for which variable the value is held (or the different variables must share the same bit representation).

2.1.25 `std::variant<>`

A type-safe standard library wrapper around union which is less error-prone.

```
std::variant<int, float> v = 42; // v holds an int
bool isInt = holds_alternative<int>(v); // true
int w = std::get<int>(v);
int w = std::get<0>(v); // same effect as the previous line
int w = std::get<float>(v); // throws std::bad_variant_access
```

2.1.26 `std::visit<>()`

Calls a functor (function object) – called the *visitor* – on the value currently contained in a variant. The functor is a class or struct whose operator() is overloaded for each type of the variant, so that it can actually be called on all possible types.

2.1.27 `enum`

A mnemonic representation of small sets of integer values, used to make code more readable and less error-prone than if we had to use the ints directly. Enumerators from a (plain) enum are entered into the same scope as the name of their enum and implicitly converts to their integer value, hence they're also referred to as unscoped enums.

```
enum Color { red, blue };
enum Traffic_light { green, yellow }; // We cannot redefine red or blue; they're in the same scope

bool sameInt = Color::red == Traffic_light::green; // true; they're both 0
Color c = 1; // Same as Color c = Color::red / Color c = red;
```

2.1.28 `enum class`

The difference with enum is that the enumerators are in the scope of their enum class instead of outside of it, so they can be used repeatedly in different enum classes without confusion, but enumerators from different enum classes – as well as enumerators and ints – can't be freely mixed anymore. Also called scoped enums.

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red }; // We can now use the same enumerator in different
     enums
```

Since an enumeration is a user-defined type, we can define operators for it:

```
Traffic_light& operator++(Traffic_light& t)//prefix increment: ++ {
   switch (t) {
      case Traffic_light::green: return t=Traffic_light::yellow;
      case Traffic_light::yellow: return t=Traffic_light::red;
      case Traffic_light::red: return t=Traffic_light::green;
   }
}
```

Using enum struct is exactly the same.

2.1.29 `std::underlying_type<>`

A templated struct which – if supplied with a complete enumeration type (either enum or enum class) – provides a typedef member called type that gives the underlying type of the enumeration type (which by default is int for scoped enumerations and some implementation specific integral type for unscoped enumerations).
If you want another underlying type for an enumeration type, you have to explicitly declare this:

```
enum class e: unsigned {}; // Uses unsigned ints as underlying type
```

2.1.30 `decltype()`

Returns the type of an expression. If given the name of a variable (an "id-expression"), it returns the type of the variable, if given an lvalue of type `T`, it returns `T&`, and if given an rvalue of type `T`, it returns `T`.

```
int n = 10;
decltype(n) a = 20; // a is an int
decltype((n)) b = a; // b is an int&, because (n) is an lvalue
```

2.2 Inheritance

2.2.1 `virtual` function

```
class SomeClass {
   public:
      virtual int& getNumber() const { return number; }; // Virtual function
   private:
      int number;
}
```

`virtual` indicates that a class deriving the current class may redefine this function, in which case this more specific implementation is called on objects of the derived class, even if those are declared as the base class. The base class must provide a definition of a `virtual` function, which serves as the default function for all derived classes that don't reimplement it.

2.2.2 Pure `virtual` function

Putting `= 0` behind a `virtual` function declaration makes it purely virtual. Having a pure virtual function makes a class abstract, meaning it can't be instantiated. A concrete deriving/inheriting class *must* (re)implement any pure virtual functions; if they don't, they become abstract as well.

```
class AbstractBase {
   public:
      virtual void function() const = 0; // Pure virtual function
   private:
      int number;
}

AbstractBase ab; // Error: there can be no objects of an abstract class
AbstractBase c = new Child(10); // OK: AbstractBase is a base class/interface for Child
```

A base class can still give a definition, however. An example of a use case for this is when there is a reasonable default behaviour, but the designer only wants this behaviour to be called explicitly, i.e. the base class's implementation must be explicitly called from the derived implementation:

```
class Child : public AbstractBase { // AbstractBase's declaration of function() would have to be
    public or protected to avoid a compile time problem

   virtual void function() {
      AbstractBase::function(); // Class Child doesn't have anything special to do for function()
          so we'll call AbstractBase's
   }

};
```

2.2.3 Concrete types

Types of which the representation is part of the definition, i.e. they have no pure `virtual` functions. Can be instantiated.

2.2.4 Abstract types

A type that completely insulates a user from implementation details. The interface is decoupled from the representation, and genuine local variables are given up. Since we don't know anything about the representation of an abstract type (not even its size), we must allocate objects on the free store and access them through references or pointers.

More concretely, abstract types are types with at least one purely `virtual` function, and such classes cannot be instantiation; they can only be inherited from by other, potentially concrete classes.

A virtual destructor is essential for an abstract class, because an object of a derived class is usually manipulated through the interface provided by its abstract base class (i.e. a pointer to the base type), and someone destroying an abstract container class through a pointer has no idea what resources are owned by this derived implementation. Then, the virtual function call mechanism ensures that the proper destructor is called. That destructor then implicitly invokes the destructors of its members and bases, so that objects are destroyed "top-down".

A class that provides the interface to a variety of other classes is often called a *polymorphic type*. Polymorphic functions (having an (abstract) base class as parameter type) needn't be recompiled if the implementation of a derived class of the parameter type changes or a brand-new class is derived. The flip side of this flexibility is that objects must be manipulated through pointers or references.

2.2.5 Inheritance

Classes in class hierarchies are different from concrete, singular classes: we tend to allocate them on the free store using `new`, and we access them through pointers or references. Two benefits are *interface inhertiance* (an object of a derived class can be used wherever an object of a base class is required; the base class acts as an interface for the derived class) and *implementation inheritance*.

```
class DerivedClass : public BaseClass { // BaseClass can be either abstract or concrete
   public:
      virtual int& getNumber() override { return number; }; // Override a function that base class
          defines as well
   private:
      int number;
}
```

2.2.6 Kinds of inheritance

The `public`/`protected`/`private` keyword after the `:` decides what code is "aware of" the inheritance:

```
class Base
{
   public:
      int x;
   protected:
      int y;
   private:
      int z;
};

class ChildA : public Base
{
   // x is public
   // y is protected
   // z is not accessible from ChildA
};

class ChildB : protected Base
{
   // x is protected
   // y is protected
   // z is not accessible from ChildB
};

class ChildC : private Base // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from ChildC
};
```

`ClassA`, `ClassB` and `ClassC` all contain the variables `x`, `y` and `z`. It is just question of access. If a concrete class is derived from another concrete class, its constructor [always?] calls the constructor of its parent class, either implicitly through a default constructor or in the initializer list in case we need to supply arguments.

2.2.7 `override`

Optional keyword used to make explicit to readers and the compilers (so that it can catch spelling errors or type differences) that the current function definition overrides the base class's definition. It's not actually needed, though.

2.2.8 Virtual function table

A mechanism for *dynamic dispatch*, i.e. resolving which function implementation to use when a function is called on an object declared as a base type. The usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions, called the virtual function table or simply the `vtbl`. Each class with virtual functions has its own `vtbl` identifying its virtual functions. The implementation of the caller needs only to know the location of the pointer to the `vtbl` in a base class and the index used for each virtual function. This virtual call mechanism can be made almost as efficient as the "normal function call" mechanism (within 25%). Its space overhead is one pointer in each object of a class with virtual functions plus one `vtbl` for each such class.

2.2.9 Class hierarchy

A set of classes ordered in a lattice created by derivation. We use class hierarchies to represent concepts that have hierarchical relationships.

2.2.10 `dynamic_cast`

Attempts to cast the runtime type of a variable. We can use this to determine whether the pointer to the base type is at runtime actually some derived type, e.g. in order to use a member function that is only provided by a particular derived class:

```
Base* pb;
// pb changes during runtime
if (Derived* pd = dynamic_cast<Derived*>(pb)) {
   // pd points to a Derived; use it
}
```

If at runtime the object pointed to by the argument of `dynamic_cast` (`pb`, a `Base*`) is not of the expected type (`Derived*`) or a class derived from the expected type, `dynamic_cast` returns `nullptr`.

When a different type is unacceptable, we can simply `dynamic_cast` to a reference type. If the object is not of the expected type, `dynamic_cast` throws a `std::bad_cast` exception:

```
Derived& r {dynamic_cast<Derived&>(*pb)}; // Catch std::bad_cast somewhere
```

2.3 Container

A `struct` or `class` that serves as a *resource handle* (a class responsible for an object accessed through a pointer) to a collection of elements elsewhere, e.g. a `class` that manages an array in dynamic memory using `new` and `delete`.

2.3.1 std::vector<>

The most common standard library container type. Its elements are stored contiguously in memory on the heap/free store. A vector holds pointers to the first element, one-past-the-last element, one-past-the-last allocated space, and an allocator responsible for acquiring and releasing memory through `new` and `delete`. `size()` is a member function giving the number of elements. Subscripting with `[]` is supported but doesn't range-check due to overhead; `at()` does the same thing but with range-checking, i.e. throwing an exception of type `out_of_range`. `push_back()` adds an element to the end of the vector. `emplace_back()` takes arguments for an element's constructor and constructs it in a newly allocated space in the container, avoiding copying it.

```
// Allocation initialization
std::vector<bool> bools(3); // Contains 3 default-constructed bools
std::vector<bool> nobools; // Contains 0 bools

// List initialization
std::vector<int> ints = {1, 2, 3, 4, 5};

// Equivalued initialization
vector<bool> vect(10, true); // Contains 10 booleans that are all true

// Declaration and loop push_back
std::vector<int> ints;
for (int i = 0; i < 10; ++i) {
   ints.push_back(i);
}

// Range-for loops are supported
void print_book(const vector<int>& ints) {
for (const auto& x : book)
   // ...
}
```

Copying or assigning a `std::vector` involves copying its elements.

`std::vector<bool>` is a special optimization, which stores each boolean in one bit instead of the usual 8. This saves a lot of memory, but makes a lot of the usual `std::vector` operation, like `operator[]` and `data()` unavailable.

Do not store objects of polymorphic type in a container directly, because specializations may increase the size; instead, use pointers.

`std::vector`s are highly efficient, and you should almost always use these over regular arrays. When doubting its performance, measure. Subscripting and traversal are cheap and easy, but insertion and deletion may involve moving all elements to another place in memory. Returning a vector is efficient because vector provides move semantics.

2.3.2 `std::list<>`

A doubly-linked list. Meant for sequences where we want to insert and delete elements without moving other elements. Useful when requiring many `insert(T* elem, T elemNew)` and `erase(T* elem)` operations.

2.3.3 `std::forward_list<>`

A singly-linked list. Allows only forward iteration, but saves space by not storing a pointer to the previous element. Doesn't even keep track of its number of elements, contrary to `std::list`.

2.3.4 `std::map<>`

A balanced binary search tree that associates, or maps, keys to values and stores those as pairs. Subscripting a map with a key type return the associated value if found, or creates a new pair with a default constructed value. You can also use `find()` and `insert()`. Lookup is $O(\log n)$.

2.3.5 `std::unordered_map<>`

Uses a hash table and doesn't require an ordering function, hence the name. It does require a hash function as second template variable. C++ provides hasing function for built-in types and `std::string`s as function objects. For user-defined types, creating a new hash function by combining existing hash functions using exclusive-or (`^`) is simple and often very effective. You can do this by specializing the standard library `hash`:

```
template<> struct hash<Record> {
   using argument_type = Record;
   using result_type = std::size_t;

   size_t operator()(const Record& r) const {
      return hash<string>()(r.name) ^ hash<int>()(r.product_code);
   }
}
```

An `unordered_map` is much faster than a `map` for large containers when using a good hash function, but worst-case performance is far worse when using a bad one.

2.3.6 `std::multimap<>`

A `std::map` in which a value can occur multiple times.

2.3.7 `std::set<>`

A `std::map` with just keys and no values, i.e. a sorted collection of unique elements.

2.3.8 `std::multiset<>`

A `std::set` in which a value can occur multiple times.

2.3.9 `std::unordered_set<>`

A `std::set` with using hashed lookup.

2.3.10 `std::deque<>`

A double-ended queue.

2.3.11 `size()`

Conventional member function of container classes for getting the number of elements it holds.

2.3.12 `begin()`

Conventional member function of container classes for getting an iterator pointing to the first element. Implementation may looks sorta like:

```
template<typename T>
T* begin(SomeContainer<T>& x) {
   return x.size() ? &x[0] : nullptr; // Pointer to first element or nullptr
}
```

2.3.13 `end()`

Conventional member function of container classes for getting an iterator pointing to the one-beyond-the-last element. Implementation may looks sorta like:

```
template<typename T>
T* end(SomeContainer<T>& x) {
   return x.size() ? &x[0] + x.size() : nullptr;
}
```

2.3.14 Iterator

Each container type has an associated iterator of type `C::iterator`. Even though their implementation may differ greatly (because the containers themselves differ), they all share the same semantics and operators, so that they may essentially be used as a pointer to a container's elements. They are used to traverse a sequence/container or to pass (sub)sequences to other functions. Should support `++` to move to the next element and `*` to access the value of the element pointed to. Iterators are concepts.

2.3.15 `std::hash<K>` and `std::unordered_map<K,V>` types

The standard-library `std::unordered_map<K,V>` is a hash table with `K` as the key type and `V` as the value type. To use a type `K` as a key, we must define `hash<K>`. The standard library does that for us for common types, such as `std::string`.

2.4 Templates

Compile-time mechanism (i.e. no run-time overhead) for parametrizing a class:

```
template<typename T> // Makes T a parameter of the declaration it prefixes
class SomeContainer {
   private:
      T* elem; // Elem points to an array of sz elements of type T
   public:
      // Constructor, destructor, etc. etc.
      T& operator[](int i);
}
```

or a member function:

```
template<typename T> // Basically means "for all types T"
SomeContainer<T>::SomeContainer(int size) {
   if (size < 0)
      throw Negative_size{};
   elem = new T[size];
}
```

We can then *instantiate/specialize* these:

```
SomeContainer<std::string> sc(17); // SomeContainer of 17 strings
```

2.4.1 `typename`

Keyword used to introduce a type parameter (in `template<typename T>`).

2.4.2 `class`

Equivalent to `typename` (e.g. `template<class T>`). Found more in older code.

2.4.3 Value template arguments

Template value arguments must be constant expressions.

```
template<typename T, int N>
```

2.4.4 Read-only access to template arguments

It is very common for a parameterized type to provide an alias for types related to their template arguments:

```
template<typename T, int N>
struct Buffer {
   using value_type = T; // Alias to get the type T
   constexpr int size() { return N; } // constexpr function to get the value N
   T[N];
}
```

Every standard-library container provides `value_type` as the name of its value type. The aliasing mechanism can be used to define a new template by binding some or all template arguments:

```
template<typename Key, typename Value>
class Map {
   // ...
};

template<typename Value>
using StringMap = Map<std::string, Value>;

StringMap<int> m; // m is a Map<std::string, int>
```

2.4.5 Default template arguments

```
template<typename T1, typename T2 = T1> // If we don't supply a second argument, the types will be
    equal
```

2.4.6 Template argument deduction (C++17)

Instead of

```
pair<int,double> p = make_pair(1,5.2); // p is a pair<int,double>
```

we can do

```
pair p = {1,5.2}; // p is a pair<int,double>
```

So we don't need to annoyingly specify the template parameters.

2.4.7 Deduction guides

In case that the compiler cannot deduce template arguments – such as when a custom `Vector` type is range-initialized with a beginning and end iterator – we can add a *deduction guide* after the constructor:

```
template<typename Iter>
Vector(Iter, Iter) -> Vector<typename Iter::value_type>
```

So now, when a `Vector` is initialized with two iterators from any container datatype (including `Vector` itself), it will automatically deduce to the iterator's value type.

2.4.8 Variable templates

```
template <class T>
constexpr T viscosity = 0.4;

template <class T>
constexpr space_vector<T> external_acceleration = { T{}, T{-9.8}, T{} };

auto vis = 2 * viscosity<double>;
auto acc = external_acceleration<float>;
```

2.4.9 Variadic templates

Allows a template to take an arbitrary number of arguments of arbitrary type. Concretely, a variadic template has at least one *parameter pack*, indicated by `...`:

```
template<typename... Ints>
void function(Ints... ints) {
  int args[] { ints... }; // Unpack ints here
}
```

For example:

```
template<typename Head, typename... Tail> // Tail is a sequence of types
void print(Head head, Tail... tail) {// tail is a sequence of values of the types in Tail
   cout << head << ' ';
   print(tail...); // Don't forget to catch for an empty parameter list
}
```

Similar to variadic arguments.

2.4.10 Fold expressions (C++17)

Shorthand notation using `...` that compounds the arguments in a variadic template. Examples:

```
template<Number... T>
int sum(T... v) {
   return (0 + ... + v); // Left fold: (((((0+v[0])+v[1])+v[2])+v[3])+v[4]). OR:
    return (v + ... + 69); // Right fold: (v[0]+(v[1]+(v[2]+(v[3]+(v[4]+69)))))
}

template<typename ...T>
void print(T&&... args) {
    (std::cout << ... << args); // Print all arguments
}

template<typename T, typename... Ts>
vector<T> to_vector(Ts&&... ts)
{
    vector<T> res;
    (res.push_back(ts) ...);
    return res;
}
```

2.4.11 `std::forward<>()`ing arguments

```
template<typename T>
void intermediary(T&& arg)
{
    destination(std::forward<T>(arg));
}
```

`std::forward<T>(arg)` forwards `arg` (which is always an lvalue) as either an lvalue or as rvalue, depending on `T`. It moves/forwards the arguments unchanged, and is essentially a more sophisticated `std::move()` by correctly handling subtleties to do with lvalue and rvalue. Use `std::forward()` exclusively for forwarding and don't `std::forward()` something twice; once you have forwarded an object, it's not yours to use anymore.

2.4.12 Template compilation

Argument checking for unconstrained template arguments is postponed until code is generated for the template and a set of template arguments, i.e. at (template) instantiation time. This late checking has two disadvantages:

- Type errors can be found uncomfortably late and with difficult error messages, because the compiler found the problem only after combining information from several places in the program.
- We have to resort to *duck typing*, i.e. we check with respect to values rather than object types.

To use an unconstrained template, its definition (not just its declaration) must be in scope at its point of use. In practice, this means that template definitions are typically found in header files, rather than `.cpp` files. This changes when we start to use modules, in which case the source code is organized in the same way for ordinary functions and template functions.

2.5 Concepts (C++20)

Constrained template arguments, or, requirements on template arguments, or, predicates that check whether template argument `T` has all the properties that whatever it prefixes requires (else it throws a compile-time error). Concepts are abstractions representing the fundamental operations and data structures (i.e. concepts) that *generic programming* aims to abstract away from and into generic algorithms applicable on a wide variety of data structures. The process of generalizing from concrete code while preserving performance is called *lifting*, and often simply entails replacing concrete types with template arguments.
Concepts help with catching errors during compile-time. For example, when we need a sequence:

```
template<Sequence T> // "For all T such that Sequence(T)"
```

which is equivalent to

```
template<typename T>
requires Sequence<T> // Requires that T is a Sequence
```

Here, `Sequence` is a predicate, or a *concept*. Other predicates are `Element`, which checks whether a type has all the properties that a vector requires of its elements.

2.5.1 Standard library concepts

A small selection:

- `Element`; a type with all properties that a vector requires of its elements.
- `Regular`; is `DefaultConstructible`, `Copyable`, `EqualityComparable`, and doesn't suffer from technical problems due to overly clever programming tricks.
- `Semiregular`; only `DefaultConstructible` and `Copyable`.
- `Copyable`; `CopyConstructable`, `Moveable` and `Assignable`.
- `StrictTotallyOrdered`; can be compared with `<`, `<=`, `>` and `>=`.
- `Range`; a sequence of element, defined by either `begin()` and `end()`, or `begin()` and a number n of elements, or `begin()` and a predicate that is `true` for the end-of-sequence. Thanks to this concept, where a standard-library algorithm requires a sequence defined by a pair of iterators, C++20 will allow a `Range` as a notationally simpler alternative (e.g. `std::sort(v)` rather than `sort(v.begin(),v.end())`).

2.5.2 `requires`-clause

The `requires`-clause specifies a requirement on the template parameter. This *constrained argument* specifies a *constrained template*. The `typename` introducer requires only that the argument is a type.

A more complete example:

```
template<Sequence Seq, Number Num> // We should define what Sequence and Number mean
requires Arithmetic<Value_type<Seq>, Num> // We need to be able to add the element type of Seq
    (Value_type<Seq>) to the type of the accumulator (Num)
Num sum(Seq s, Num v) {
   for (const auto& x : s) v+=x;
   return v;
}
```

This can be written more verbose:

```
template<typename Seq, typename Num>
requires Sequence<Seq> && Number<Num> && requires Arithmetic<Value_type<Seq>, Num>
Num sum(Seq s, Num n);
```

or even less verbose:

```
template<Sequence Seq, Arithmetic<Value_type<Seq>> Num>
Num sum(Seq s, Num n); // Num is a patially specified requirement
```

2.5.3 `requires`-expression

Used within a `requires`-clause, yielding `true` if the statements in it are valid code and `false` if they're not.

```
template<Forward_iterator Iter, int n>
requires requires(Iter p, int i) { p[i]; p+i; } // Iter has subscripting and addition
void advance(Iter p, int n) // move p n elements forward
{
   p += n; // A random-access iterator has +=
}
```

However, if you see `requires requires` in your code, it is probably too low level; `requires`-expressions are the assembly code of generic programming. Prefer use of properly named concepts with well-specified semantics, and use `requires`-expressions in the definition of those.

2.5.4 Concept-based overloading

We can overload function that have the same type (i.e. same arguments and return type) based only on their template concept. This means that we have different function bodies for different template concepts. The compiler will select the template with the strongest requirements met by the template arguments. Like regular overloading, this is also a compile-time mechanism.

2.5.5 Defining concepts

Besides the concepts provided in the standard library and other libraries, we can define our own. For example, if we want to define a concept for types that are equality comparable:

```
template<typename T1, typename T2 = T1> // Default template argument
concept Equality_comparable = requires (T1 a, T2 b) {
   { a == b } -> bool; // Compare a T1 to a T2 with ==
   { a != b } -> bool; // Compare a T1 to a T2 with !=
   { b == a } -> bool; // Compare a T2 to a T1 with ==
   { b != a } -> bool; // Compare a T2 to a T1 with !=
};
```

For each line in the `requires`-body, the expression inside the `{ }` must be correct and must yield the type behind the `->`. Using default template arguments, we only need to supply one type. If we only supported comparing the same type, the `requires`-body would be only two lines.

We can test using `static_assert`:

```
static_assert(Equality_comparable<int,string>); // Fails
```

A more complicated example:

```
template<typename S>
concept Sequence = requires(S a) {
   typename Value_type<S>; // S must have a value type.
   typename Iterator_type<S>; // S must have an iterator type.
   { begin(a) } -> Iterator_type<S>; // begin(a) must return an iterator
   { end(a) } -> Iterator_type<S>; // end(a) must return an iterator
   requires Same_type<Value_type<S>,Value_type<Iterator_type<S>>>;
   requires Input_iterator<Iterator_type<S>>;
};
```

3 Functions

3.1 Function templates

```
template<typename Sequence, typename Value>
Value sum(const Sequence& s, Value v) { // v is initial value of the accumulator
   for (auto x : s) v += x;
   return v;
}

std::list<double>& ld;
// ...
double sum = sum(ld,0.0); // Types of the template arguments are deduced from function arguments
```

Member functions can be function templates, but a `virtual` members cannot. (The compiler would not know all instantiations of such a template in a program, so it could not generate a `vtbl`).

3.2 Function objects/functors

Also called *functors*. Objects that can be called like functions, implemented as a class with a function call operator `operator()`:

```
template<typename T>
class LessThan {
   const T val; // Value to compare against
   public:
      LessThan(const T\& v) : val{v} { } // Constructor
      bool operator()(const T\& x) const { return x < val; } // Call operator
}

LessThan lti {42}; // lti(i) will compare i to 42. Template argument is deduced
LessThan<std::string> lts {"Naur"}; // "Naur" is a C-style string, so we need <std::string> to get
    the right <
```

```
bool b1 = lti(12); // True
bool b2 = lts("Zzz"s); // False
```

Function objects are widely used as arguments to algorithms to serve as predicates.

3.3 Lambda expressions

Defines an unnamed function at the place of its use with notation `[]{ }`, e.g. `[&](int a){ return a<x; }`. The `[&]` is a capture list specifying that all local names (from outside the lambda) used in the lambda body (such as `x`) will be accessed through references. If you want to reference only `x`, you can specify this using `[&x]`. For copying `x`: `[=x]`. Capturing nothing is `[]`, capture all local names used by reference is `[&]`, and capture all local names used by value is `[=]`.

A *generic lambda* is essentially a templated lambda; a lambda with an `auto` parameter, meaning that any type is accepted as an initializer. (For reasons lost in standards committee politics, this use of `auto` is not currently allowed for function arguments.)

3.4 Variadic arguments

Allows a function to accept any number of (extra) argument, indicated by a (trailing) ... following a (possibly empty) parameter list and an optional comma:

```
void function(int arg, ...); \\ The comma is optional, but makes it more readable
void function(int arg...); \\ This is the exact same
void function(...); \\ This one doesn't necessarily first take an int
```

3.5 Standard library functions

3.5.1 `std::swap<>()`

The standard-library provides a `std::swap<T>(T& a, T& b)` implemented as three move operations: `tmp = a, a = b, b = tmp`. Many algorithms make use of a class's `swap()` function and assume that it is very fast and doesn't throw an exception. Implement such a function for types that are expensive to copy and could plausibly be swapped (e.g., by a sort function) and/or give it move operators.

4 Strings and regex

4.1 Strings

4.1.1 C-style string

A zero-terminated array of `char`.

4.1.2 `std::string`

A standard library wrapper type around C-style strings. It supports a lot of operations, like concatenation using the `+` (and `+=`) operator, indexing (using `[]` or `at()`), `substr()`, `replace()`, access to the C-style string using `c_str()`, etc. Has a move constructor, so returning even long strings by value is efficient.
Implementation-wise, short strings are kept in the `string` object itself and only longer strings (implementation-dependent, but approx. 14 characters or longer) are placed on free store via a pointer. This representation adapts accordingly if the string value changes.
`std::string` is actually an alias: `using string = basic_string<char>;`. If we have another character set, we can instantiate our own `basic_string`.

4.1.3 `std::string_view`

A read-only (pointer, length) pair denoting a sequence of characters that are not owned by the `string_view` itself. Useful as an abstraction of specific string implementations and for passing substrings to functions.

4.1.4 Raw string literals

Use a *raw string literal* when you want backslashes and quotes directly in the string. Replace the double quotes by `R"(` and `)"`. Useful for regex.

4.2 Regex

Regex (short for *regular expressions*) are a powerful tool for pattern matching and searching strings.

4.2.1 Variants/notations

The `regex` library can recognize several variants/notation for regex. EMCA (used in ECMAScript aka JavaScript) is the default and is treated here.

4.2.2 Newline

`\n` can be used to search multiline patterns.

4.2.3 Syntax

- `.`: Any single character (a "wildcard")
- `\\`: Next character has a special meaning
- `|`: Alternative (or)
- `^`: Start of line; negation
- `\$`: End of line
- `[  `: Character class
- `( )`: Grouping

4.2.4 Groups/subpatterns

A *group*, or subpattern, is enclosed in parantheses: `( )` and are extracted separately. If you need parentheses that don't define a subpattern, use `(?` rather than `(`. Groups don't nest.

4.2.5 Repetition

- `{ }`-suffix: Repetition count, i.e.:
  - `{n}`-suffix: Exactly `n` times
  - `{n,}`-suffix: `n` or more times
  - `{n,m}`-suffix: At least `n` and at most `m` times
- `*`-suffix: Zero or more (suffix operation)
- `+`-suffix: One or more (suffix operation)
- `?`-suffix: Optional (zero or one) (suffix operation)

4.2.6 Lazy pattern matching

By default, the pattern matcher always looks for the longest match, known as the Max Munch rule. A suffix `?` after any of the repetition notations (`?`, `*`, `+`, and `{ }`) makes the pattern matcher *lazy*, or "non-greedy", i.e. it will look for the shortest match, e.g. `(ab)+` matches all of `ababab`, but `(ab)+?` matches only the first `ab`.

4.2.7 Character classes

A character class is defined by a `[ ]` pair, within which a character class name must be further bracketed by `[: :]`. Common character class names are:

- `alnum`: Any alphanumeric character
- `alpha`: Any alphabetic character
- `blank`: Any whitespace character that is not a line separator
- `cntrl`: Any control character
- `digit`/`d` Any decimal digit
- `graph`: Any graphical character
- `lower`: Any lowercase character
- `print`: Any printable character
- `punct`: Any punctuation character
- Any whitespace character
- `space`/`s`: Any whitespace character (space, tab, etc.)
- `upper`: Any uppercase character
- `w`: Any word character (alphanumeric characters plus the underscore)
- `xdigit`: Any hexadecimal digit character

There are direct abbreviations for some of these:

- `\d` abbreviates `[[:digit:]]`
- `\D` abbreviates `[^[:digit:]]`
- `\s` abbreviates `[[:space:]]`
- `\S` abbreviates `[^[:space:]]`
- `\w` abbreviates `[_[:alnum:]]`
- `\W` abbreviates `[^_[:alnum:]]`
- `\l` abbreviates `[[:lower:]]`
- `\L` abbreviates `[^[:lower:]]`
- `\u` abbreviates `[[:upper:]]`
- `\U` abbreviates `[^[:upper:]]`

4.2.8 `std::regex_match()`

`regex_search(string, pattern)` tests whether `string` is an identifier.

4.2.9 `std::regex_search()`

`regex_search(string, matches, pattern)` searches `string` for anything that matches the regular expression stored in `pattern`, returning found matches through `matches`, which is of type `smatch`; a vector of (sub)patterns of type `std::string`. The first elemet at index `0` is the complete match. If no matches are found, `false` is returned.

4.2.10 `std::regex_iterator<>()`

A bidirectional iterator, useful for iterating pattern matches in a character sequences. The standard library defines

```
using sregex_iterator = regex_iterator<string>;
```

which we can use to iterate through all matches like this:

```
for (sregex_iterator p(string.begin(), string.end(), pattern);
   p != sregex_iterator{};
   ++p) {
      std::cout << (*p)[1] << '\n';
   }
```

The default `regex_iterator{}` is the only possible end-of-sequence. These iterators cannot be used to write through.

5 IO

5.1 Streams

In `<istream>` and `<ostream>`, the IO stream library defines formatted (meaning bytes are grouped and sent as types such as `int`s, whereas unformatted streams send raw bytes) output respectively input for every built-in type. Both use character string representations of built-in types and can easily be extended for user-defined types. `<fstream>` and `<sstream>` provide streams to and from a file and a `std::string` respectively. `<iomanip>` provides (parametrized) manipulators. The stream classes are written as template classes in order to support various character sets.

`ios` is the topmost class in the streams hierarchy, and is inherited by `istream`, `ostream`, and `streambuf`. `istream` and `ostream` are inherited by `iostream`, which combines the two. `istringstream` and `ifstream` both inherit from `istream` and `ostringstream` and `ofstream` both inherit from `ostream`.

5.1.1 `std::ostream`

An `std::ostream` converts typed objects to a stream of byte characters.

5.1.2 `std::istream`

An `std::istream` converts a stream of byte characters to typed objects, and takes care of memory management and range checking.

5.1.3 `std::iostream`

Inherits from both `std::istream` and `std::ostream` and holds state information, such as formatting information, error state (with `iostream::fail()`), whether end of input has been reached (with `iostream::eof()`) and what kind of buffering is used.

5.1.4 `std::ofstream`

An `std::ostream` for writing to a file. Constructor takes a `std::string` filepath as argument. Can be used as an ordinary `std::ostream` (just like `std::cout`), but maybe first check whether the file was properly opened by checking the `std::ofstream` object itself.

5.1.5 `std::ifstream`

An `std::istream` for reading from a file. Constructor takes a `std::string` filepath as argument. Can be used as an ordinary `std::istream` (just like `std::cin`), but maybe first check whether the file was properly opened by checking the `std::ifstream` object itself.

5.1.6 `std::fstream`

An `std::iostream` [?] for writing to and reading from a file. Constructor takes a `std::string` filepath as argument.

5.1.7 `std::ostringstream`

An `std::ostream` for writing to a `std::string`. Constructor requires no arguments. Can be used as an ordinary `std::ostream` (just like `std::cout`). The result can be obtained with `ostringstream::str()`. Commonly used to format before giving the resulting string to a GUI.

5.1.8 `std::istringstream`

An `std::istream` for reading from a `std::string`. Constructor requires no arguments. Can be used as an ordinary `std::istream` (just like `std::cin`). Commonly used to put strings from a GUI read using formatted input operations into.

5.1.9 `std::stringstream`

An `std::iostream` [?] for reading from and writing to a `std::string`.

5.1.10 `<<`

The "put to" operator `<<` writes its right operand into the left (which are objects of type `ostream`) and returns the left. Because it returns the left operands, multiple values (of potentially different type) can be output by chaining `<<` operations together.
User-defined classes or structs can overload `operator<<()` takes its output stream (by reference) as its first argument, a reference to its own type as second argument, and returns it as its result, e.g.:

```
ostream& operator<<(ostream& os, const SomeClass& cl) {
   return os << "{\"" << cl.name << "\", " << cl.number << "}";
}
```

5.1.11 `>>`

The "get from" operator `>>` reads left operand into the right and returns the left. The type of the right-hand operand determines the type of input that is accepted. Input operations can also be chained. By default, `>>` skips initial whitespace (e.g. space or newline).
A `std::string` is a convenient output operand for reading a sequence of characters. By default, whitespace terminates the read.
`for (int i; is >> i;){ vec.push_back(i); }` reads integers into `i` until a non-integer is encountered, because `is >> i` returns (a reference to) `is`, which is a `std::iostream` and which tests to `true` if the last read succeeded and the stream is ready for another operation.

5.1.12 `std::getline()`

`std::getline(istream, std::string)` can be used to read a whole line, terminated by a newline, into the `istream` (e.g. `std::cin`).

5.1.13 `std::cout`

A `std::ostream` that is the standard output stream. By default, values written to `std::cout` are converted to a sequence of characters, i.e. `std::cout << 123;` (and also `int x = 123; std::cout << x;`) first places the character `1` and lastly places the character `3` into the stream.

5.1.14 `std::cerr`

`std::cout` is the standard output stream for reporting errors.

5.1.15 `std::cin`

A `std::istream` that is the standard input stream.

5.1.16 Manipulators

Formatting controls that you write to an `std::ostream` to determine its formatting:

```
std::cout << 123         // Decimal format: 123
          << hex << 123  // Hexadecimal format: 4d2
          << oct << 123; // Octal format: 2322

constexpr double d = 123.456;

std::cout << d              // Default notation: 123.456
          << scientific << d // Scientific notation: 1.234560e+002
```

```
        << hexfloat << d // Hexadecimal notation: 0x1.edd2f2p+6
        << fixed << d    // Fixed notation: 123.456000
        << defaultfloat << d // Default again: 123.456
```

The number of total digits of the default notation, and the number of digits after the decimal of the scientific and fixed notations are defined by the precision integer.

5.1.17 `ostream::precision()`

Takes an `int` and sets the precision for its `std::ostream` from then on. Number are rounded rather than truncated. Doesn't affect integer output.

5.1.18 Stream iterators

Iterators for streams, most often used as arguments to algorithms.

```
ostream_iterator<string> oi {std::cout}; // You can write to *oi just like you would to std::cout
istream_iterator<string> ii {cin}; // Treat std::cin as a read-only container
istream_iterator<string> eos {}; // The default istream_iterator indicates the end of input. Use
    together with ii to indicate a sequence
```

5.2 C-style IO

If you *don't* use C-style IO and care about IO performance, call `ios_base::sync_with_stdio(false);`, which avoid overhead needed for C-style compatible IO.

5.2.1 `printf()`

5.2.2 `scanf()`

5.3 Filesystem

`<filesystem>` offers a uniform interface to most facilities of most file systems.

5.3.1 `std::filesystem::path`

Type representing a filepath. Can be constructed with or implicitly converted from a `std::string` (ideally containing a filepath). It's `value_type` is the character type used by the native encoding of the filesystem (`char` on POSIX, `wchar_t` on Windows). Its `string_type` is `std::basic_string<value_type>`. The `/=`-operator concatenates two `path`s using the path separator (`/` by default), and the `+=`-operator concatenates without the separator. `path::stem()` returns the stem part, `path::filename()` returns the filename part, and `path::extension()` returns the file extension part of a path.

5.3.2 `std::filesystem::filesystem_error`

A file system exception.

5.3.3 `std::filesystem::directory_entry`

A directory entry, used by a `std::filesystem::directory_iterator`. Implicitly converts to a `std::filesystem::path`.

5.3.4 `std::filesystem::directory_iterator`

A directory iterator.

```
for (const directory_entry& x : directory_iterator{p}){ std::cout << x.path(); }
```

5.3.5 `std::filesystem::recursive_directory_iterator`

A recursive directory iterator.

5.3.6 `std::filesystem::exists()`

Check whether the file indicated by the supplied path exists.

5.3.7 `std::filesystem::is_directory(f()`

Check whether the path leads to a directory.

5.3.8 `std::filesystem::is_regular_file(f()`

Check whether the path leads to an ordinary file.

5.3.9 `std::filesystem::file_size(f()`

Get file size.

5.3.10 `std::filesystem::current_path()`

Returns the current working directory as a `path`.