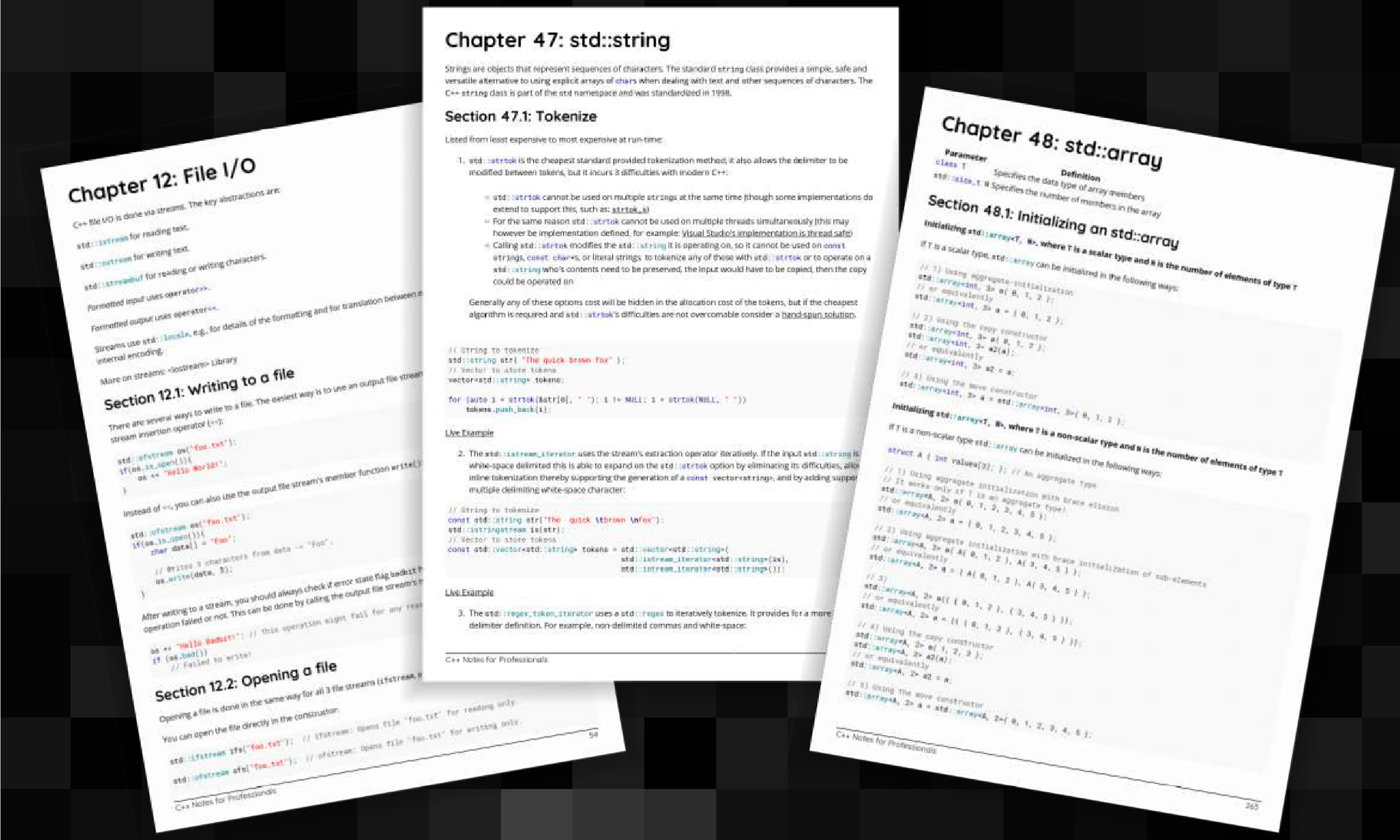
C++

Notes for Professionals



Chapter 12: File I/O

C++ file I/O is done via streams. The key abstractions are:

std::istream for reading text.

std::ostream for writing text.

std::streambuf for reading or writing characters.

Formatted input uses operator>>.

Formatted output uses operator<<.

Streams use std::locale, e.g., for details of the formatting and for translation between e
internal encoding.

More on streams: <iostream> Library

Section 12.1: Writing to a file

There are several ways to write to a file. The easiest way is to use an output file stream
stream insertion operator (<<):

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

Instead of <<, you can also use the output file stream's member function write()

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";

    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

After writing to a stream, you should always check if error state flag badbit
operation failed or not. This can be done by calling the output file stream's

```
os << "Hello Badbit!"; // This operation might fail for any rea
if (os.bad())
    // Failed to write!
```

Section 12.2: Opening a file

Opening a file is done in the same way for all 3 file streams (ifstream,

You can open the file directly in the constructor:

```
std::ifstream ifs("foo.txt");  // ifstream: Opens file "foo.txt" for reading only.
std::ofstream ofs("foo.txt");  // ofstream: Opens file "foo.txt" for writing only.
```

C++ Notes for Professionals 54

Chapter 47: std::string

Strings are objects that represent sequences of characters. The standard string class provides a simple, safe and versatile alternative to using explicit arrays of chars when dealing with text and other sequences of characters. The C++ string class is part of the std namespace and was standardized in 1998.

Section 47.1: Tokenize

Listed from least expensive to most expensive at run-time:

1. std::strtok is the cheapest standard provided tokenization method, it also allows the delimiter to be modified between tokens, but it incurs 3 difficulties with modern C++:

   - std::strtok cannot be used on multiple strings at the same time (though some implementations do extend to support this, such as: strtok_s)
   - For the same reason std::strtok cannot be used on multiple threads simultaneously (this may however be implementation defined, for example: Visual Studio's implementation is thread safe)
   - Calling std::strtok modifies the std::string it is operating on, so it cannot be used on const strings, const char*s, or literal strings. to tokenize any of these with std::strtok or to operate on a std::string who's contents need to be preserved, the input would have to be copied, then the copy could be operated on

   Generally any of these options cost will be hidden in the allocation cost of the tokens, but if the cheapest algorithm is required and std::strtok's difficulties are not overcomable consider a hand-spun solution.

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Live Example

2. The std::istream_iterator uses the stream's extraction operator iteratively. If the input std::string is white-space delimited this is able to expand on the std::strtok option by eliminating its difficulties, allo inline tokenization thereby supporting the generation of a const vector<string>, and by adding suppor multiple delimiting white-space character:

```
// String to tokenize
const std::string str("The  quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
                                        std::istream_iterator<std::string>(is),
                                        std::istream_iterator<std::string>());
```

Live Example

3. The std::regex_token_iterator uses a std::regex to iteratively tokenize. It provides for a more delimiter definition. For example, non-delimited commas and white-space:

C++ Notes for Professionals

Chapter 48: std::array

Parameter	Definition
class T	Specifies the data type of array members
std::size_t N	Specifies the number of members in the array

Section 48.1: Initializing an std::array

Initializing std::array<T, N>, where T is a scalar type and N is the number of elements of type T

If T is a scalar type, std::array can be initialized in the following ways:

```
// 1) Using aggregate-initialization
std::array<int, 3> a{ 0, 1, 2 };
// or equivalently
std::array<int, 3> a = { 0, 1, 2 };

// 2) Using the copy constructor
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> a2(a);
// or equivalently
std::array<int, 3> a2 = a;

// 3) Using the move constructor
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Initializing std::array<T, N>, where T is a non-scalar type and N is the number of elements of type T

If T is a non-scalar type std::array can be initialized in the following ways:

```
struct A { int values[3]; }; // An aggregate type

// 1) Using aggregate initialization with brace elision
// It works only if T is an aggregate type!
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
// or equivalently
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };

// 2) Using aggregate initialization with brace initialization of sub-elements
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// or equivalently
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };

// 3)
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// or equivalently
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};

// 4) Using the copy constructor
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// or equivalently
std::array<A, 2> a2 = a;

// 5) Using the move constructor
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

C++ Notes for Professionals 265

600+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is
not affiliated with official C++ group(s) or company(s).
All trademarks and registered trademarks are
the property of their respective owners

Contents

About ... 1

Chapter 1: Getting started with C++ ... 2

Section 1.1: Hello World ... 2

Section 1.2: Comments ... 3

Section 1.3: The standard C++ compilation process ... 5

Section 1.4: Function ... 5

Section 1.5: Visibility of function prototypes and declarations ... 8

Section 1.6: Preprocessor ... 9

Chapter 2: Literals ... 11

Section 2.1: this ... 11

Section 2.2: Integer literal ... 11

Section 2.3: true ... 12

Section 2.4: false ... 13

Section 2.5: nullptr ... 13

Chapter 3: operator precedence ... 14

Section 3.1: Logical && and || operators: short-circuit ... 14

Section 3.2: Unary Operators ... 15

Section 3.3: Arithmetic operators ... 15

Section 3.4: Logical AND and OR operators ... 16

Chapter 4: Floating Point Arithmetic ... 17

Section 4.1: Floating Point Numbers are Weird ... 17

Chapter 5: Bit Operators ... 18

Section 5.1: | - bitwise OR ... 18

Section 5.2: ^ - bitwise XOR (exclusive OR) ... 18

Section 5.3: & - bitwise AND ... 20

Section 5.4: << - left shift ... 20

Section 5.5: >> - right shift ... 21

Chapter 6: Bit Manipulation ... 23

Section 6.1: Remove rightmost set bit ... 23

Section 6.2: Set all bits ... 23

Section 6.3: Toggling a bit ... 23

Section 6.4: Checking a bit ... 23

Section 6.5: Counting bits set ... 24

Section 6.6: Check if an integer is a power of 2 ... 25

Section 6.7: Setting a bit ... 25

Section 6.8: Clearing a bit ... 25

Section 6.9: Changing the nth bit to x ... 25

Section 6.10: Bit Manipulation Application: Small to Capital Letter ... 26

Chapter 7: Bit fields ... 27

Section 7.1: Declaration and Usage ... 27

Chapter 8: Arrays ... 28

Section 8.1: Array initialization ... 28

Section 8.2: A fixed size raw array matrix (that is, a 2D raw array) ... 29

Section 8.3: Dynamically sized raw array ... 29

Section 8.4: Array size: type safe at compile time ... 30

Section 8.5: Expanding dynamic size array by using std::vector ... 31

Section 8.6: A dynamic size matrix using std::vector for storage 32
Chapter 9: Iterators 35
Section 9.1: Overview 35
Section 9.2: Vector Iterator 38
Section 9.3: Map Iterator 38
Section 9.4: Reverse Iterators 39
Section 9.5: Stream Iterators 40
Section 9.6: C Iterators (Pointers) 40
Section 9.7: Write your own generator-backed iterator 41
Chapter 10: Basic input/output in c++ 43
Section 10.1: user input and standard output 43
Chapter 11: Loops 44
Section 11.1: Range-Based For 44
Section 11.2: For loop 46
Section 11.3: While loop 48
Section 11.4: Do-while loop 49
Section 11.5: Loop Control statements : Break and Continue 50
Section 11.6: Declaration of variables in conditions 51
Section 11.7: Range-for over a sub-range 52
Chapter 12: File I/O 54
Section 12.1: Writing to a file 54
Section 12.2: Opening a file 54
Section 12.3: Reading from a file 55
Section 12.4: Opening modes 57
Section 12.5: Reading an ASCII file into a std::string 58
Section 12.6: Writing files with non-standard locale settings 59
Section 12.7: Checking end of file inside a loop condition, bad practice? 60
Section 12.8: Flushing a stream 61
Section 12.9: Reading a file into a container 61
Section 12.10: Copying a file 62
Section 12.11: Closing a file 62
Section 12.12: Reading a `struct` from a formatted text file 63
Chapter 13: C++ Streams 65
Section 13.1: String streams 65
Section 13.2: Printing collections with iostream 66
Chapter 14: Stream manipulators 68
Section 14.1: Stream manipulators 68
Section 14.2: Output stream manipulators 73
Section 14.3: Input stream manipulators 75
Chapter 15: Flow Control 77
Section 15.1: case 77
Section 15.2: switch 77
Section 15.3: catch 77
Section 15.4: throw 78
Section 15.5: default 79
Section 15.6: try 79
Section 15.7: if 79
Section 15.8: else 80
Section 15.9: Conditional Structures: if, if..else 80

Section 15.10: goto 81
Section 15.11: Jump statements : break, continue, goto, exit 81
Section 15.12: return 84
Chapter 16: Metaprogramming 86
Section 16.1: Calculating Factorials 86
Section 16.2: Iterating over a parameter pack 88
Section 16.3: Iterating with std::integer_sequence 89
Section 16.4: Tag Dispatching 90
Section 16.5: Detect Whether Expression is Valid 90
Section 16.6: If-then-else 92
Section 16.7: Manual distinction of types when given any type T 92
Section 16.8: Calculating power with C++11 (and higher) 93
Section 16.9: Generic Min/Max with variable argument count 94
Chapter 17: const keyword 95
Section 17.1: Avoiding duplication of code in const and non-const getter methods 95
Section 17.2: Const member functions 96
Section 17.3: Const local variables 97
Section 17.4: Const pointers 97
Chapter 18: mutable keyword 99
Section 18.1: mutable lambdas 99
Section 18.2: non-static class member modifier 99
Chapter 19: Friend keyword 101
Section 19.1: Friend function 101
Section 19.2: Friend method 102
Section 19.3: Friend class 102
Chapter 20: Type Keywords 104
Section 20.1: class 104
Section 20.2: enum 105
Section 20.3: struct 106
Section 20.4: union 106
Chapter 21: Basic Type Keywords 108
Section 21.1: char 108
Section 21.2: char16_t 108
Section 21.3: char32_t 108
Section 21.4: int 108
Section 21.5: void 108
Section 21.6: wchar_t 109
Section 21.7: float 109
Section 21.8: double 109
Section 21.9: long 109
Section 21.10: short 110
Section 21.11: bool 110
Chapter 22: Variable Declaration Keywords 111
Section 22.1: decltype 111
Section 22.2: const 111
Section 22.3: volatile 112
Section 22.4: signed 112
Section 22.5: unsigned 112
Chapter 23: Keywords 114

Section 23.1: asm ... 114
Section 23.2: Different keywords ... 114
Section 23.3: typename ... 118
Section 23.4: explicit ... 119
Section 23.5: sizeof ... 119
Section 23.6: noexcept ... 120
Chapter 24: Returning several values from a function ... 122
Section 24.1: Using std::tuple ... 122
Section 24.2: Structured Bindings ... 123
Section 24.3: Using struct ... 124
Section 24.4: Using Output Parameters ... 125
Section 24.5: Using a Function Object Consumer ... 126
Section 24.6: Using std::pair ... 127
Section 24.7: Using std::array ... 127
Section 24.8: Using Output Iterator ... 127
Section 24.9: Using std::vector ... 128
Chapter 25: Polymorphism ... 129
Section 25.1: Define polymorphic classes ... 129
Section 25.2: Safe downcasting ... 130
Section 25.3: Polymorphism & Destructors ... 131
Chapter 26: References ... 133
Section 26.1: Defining a reference ... 133
Chapter 27: Value and Reference Semantics ... 134
Section 27.1: Definitions ... 134
Section 27.2: Deep copying and move support ... 134
Chapter 28: C++ function "call by value" vs. "call by reference" ... 138
Section 28.1: Call by value ... 138
Chapter 29: Copying vs Assignment ... 140
Section 29.1: Assignment Operator ... 140
Section 29.2: Copy Constructor ... 140
Section 29.3: Copy Constructor Vs Assignment Constructor ... 141
Chapter 30: Pointers ... 143
Section 30.1: Pointer Operations ... 143
Section 30.2: Pointer basics ... 143
Section 30.3: Pointer Arithmetic ... 145
Chapter 31: Pointers to members ... 147
Section 31.1: Pointers to static member functions ... 147
Section 31.2: Pointers to member functions ... 147
Section 31.3: Pointers to member variables ... 148
Section 31.4: Pointers to static member variables ... 148
Chapter 32: The This Pointer ... 150
Section 32.1: this Pointer ... 150
Section 32.2: Using the this Pointer to Access Member Data ... 152
Section 32.3: Using the this Pointer to Differentiate Between Member Data and Parameters ... 152
Section 32.4: this Pointer CV-Qualifiers ... 153
Section 32.5: this Pointer Ref-Qualifiers ... 156
Chapter 33: Smart Pointers ... 158
Section 33.1: Unique ownership (std::unique_ptr) ... 158
Section 33.2: Sharing ownership (std::shared_ptr) ... 159

Section 33.3: Sharing with temporary ownership (std::weak_ptr) 161
Section 33.4: Using custom deleters to create a wrapper to a C interface 163
Section 33.5: Unique ownership without move semantics (auto_ptr) 164
Section 33.6: Casting std::shared_ptr pointers 166
Section 33.7: Writing a smart pointer: value_ptr 166
Section 33.8: Getting a shared_ptr referring to this 168
Chapter 34: Classes/Structures 170
Section 34.1: Class basics 170
Section 34.2: Final classes and structs 170
Section 34.3: Access specifiers 171
Section 34.4: Inheritance 172
Section 34.5: Friendship 174
Section 34.6: Virtual Inheritance 175
Section 34.7: Private inheritance: restricting base class interface 176
Section 34.8: Accessing class members 177
Section 34.9: Member Types and Aliases 178
Section 34.10: Nested Classes/Structures 182
Section 34.11: Unnamed struct/class 186
Section 34.12: Static class members 187
Section 34.13: Multiple Inheritance 191
Section 34.14: Non-static member functions 192
Chapter 35: Function Overloading 195
Section 35.1: What is Function Overloading? 195
Section 35.2: Return Type in Function Overloading 196
Section 35.3: Member Function cv-qualifier Overloading 196
Chapter 36: Operator Overloading 199
Section 36.1: Arithmetic operators 199
Section 36.2: Array subscript operator 200
Section 36.3: Conversion operators 201
Section 36.4: Complex Numbers Revisited 202
Section 36.5: Named operators 206
Section 36.6: Unary operators 208
Section 36.7: Comparison operators 209
Section 36.8: Assignment operator 210
Section 36.9: Function call operator 211
Section 36.10: Bitwise NOT operator 211
Section 36.11: Bit shift operators for I/O 212
Chapter 37: Function Template Overloading 213
Section 37.1: What is a valid function template overloading? 213
Chapter 38: Virtual Member Functions 214
Section 38.1: Final virtual functions 214
Section 38.2: Using override with virtual in C++11 and later 214
Section 38.3: Virtual vs non-virtual member functions 215
Section 38.4: Behaviour of virtual functions in constructors and destructors 216
Section 38.5: Pure virtual functions 217
Chapter 39: Inline functions 220
Section 39.1: Non-member inline function definition 220
Section 39.2: Member inline functions 220
Section 39.3: What is function inlining? 220
Section 39.4: Non-member inline function declaration 221

Chapter 40: Special Member Functions ... 222
Section 40.1: Default Constructor ... 222
Section 40.2: Destructor ... 224
Section 40.3: Copy and swap ... 225
Section 40.4: Implicit Move and Copy ... 227
Chapter 41: Non-Static Member Functions ... 228
Section 41.1: Non-static Member Functions ... 228
Section 41.2: Encapsulation ... 229
Section 41.3: Name Hiding & Importing ... 229
Section 41.4: Virtual Member Functions ... 231
Section 41.5: Const Correctness ... 233
Chapter 42: Constant class member functions ... 235
Section 42.1: constant member function ... 235
Chapter 43: C++ Containers ... 236
Section 43.1: C++ Containers Flowchart ... 236
Chapter 44: Namespaces ... 237
Section 44.1: What are namespaces? ... 237
Section 44.2: Argument Dependent Lookup ... 238
Section 44.3: Extending namespaces ... 239
Section 44.4: Using directive ... 239
Section 44.5: Making namespaces ... 240
Section 44.6: Unnamed/anonymous namespaces ... 241
Section 44.7: Compact nested namespaces ... 241
Section 44.8: Namespace alias ... 241
Section 44.9: Inline namespace ... 242
Section 44.10: Aliasing a long namespace ... 244
Section 44.11: Alias Declaration scope ... 244
Chapter 45: Header Files ... 246
Section 45.1: Basic Example ... 246
Section 45.2: Templates in Header Files ... 247
Chapter 46: Using declaration ... 248
Section 46.1: Importing names individually from a namespace ... 248
Section 46.2: Redeclaring members from a base class to avoid name hiding ... 248
Section 46.3: Inheriting constructors ... 248
Chapter 47: std::string ... 250
Section 47.1: Tokenize ... 250
Section 47.2: Conversion to (const) char* ... 251
Section 47.3: Using the std::string_view class ... 251
Section 47.4: Conversion to std::wstring ... 252
Section 47.5: Lexicographical comparison ... 253
Section 47.6: Trimming characters at start/end ... 254
Section 47.7: String replacement ... 255
Section 47.8: Converting to std::string ... 256
Section 47.9: Splitting ... 257
Section 47.10: Accessing a character ... 258
Section 47.11: Checking if a string is a prefix of another ... 258
Section 47.12: Looping through each character ... 259
Section 47.13: Conversion to integers/floating point types ... 259
Section 47.14: Concatenation ... 260

Section 47.15: Converting between character encodings 261
Section 47.16: Finding character(s) in a string 262
Chapter 48: std::array 263
Section 48.1: Initializing an std::array 263
Section 48.2: Element access 264
Section 48.3: Iterating through the Array 266
Section 48.4: Checking size of the Array 266
Section 48.5: Changing all array elements at once 266
Chapter 49: std::vector 267
Section 49.1: Accessing Elements 267
Section 49.2: Initializing a std::vector 269
Section 49.3: Deleting Elements 270
Section 49.4: Iterating Over std::vector 272
Section 49.5: vector<bool>: The Exception To So Many, So Many Rules 274
Section 49.6: Inserting Elements 275
Section 49.7: Using std::vector as a C array 276
Section 49.8: Finding an Element in std::vector 277
Section 49.9: Concatenating Vectors 278
Section 49.10: Matrices Using Vectors 279
Section 49.11: Using a Sorted Vector for Fast Element Lookup 280
Section 49.12: Reducing the Capacity of a Vector 281
Section 49.13: Vector size and capacity 281
Section 49.14: Iterator/Pointer Invalidation 283
Section 49.15: Find max and min Element and Respective Index in a Vector 284
Section 49.16: Converting an array to std::vector 284
Section 49.17: Functions Returning Large Vectors 285
Chapter 50: std::map 287
Section 50.1: Accessing elements 287
Section 50.2: Inserting elements 288
Section 50.3: Searching in std::map or in std::multimap 289
Section 50.4: Initializing a std::map or std::multimap 290
Section 50.5: Checking number of elements 291
Section 50.6: Types of Maps 291
Section 50.7: Deleting elements 292
Section 50.8: Iterating over std::map or std::multimap 293
Section 50.9: Creating std::map with user-defined types as key 293
Chapter 51: std::optional 295
Section 51.1: Using optionals to represent the absence of a value 295
Section 51.2: optional as return value 295
Section 51.3: value_or 296
Section 51.4: Introduction 296
Section 51.5: Using optionals to represent the failure of a function 297
Chapter 52: std::function: To wrap any element that is callable 299
Section 52.1: Simple usage 299
Section 52.2: std::function used with std::bind 299
Section 52.3: Binding std::function to a different callable types 300
Section 52.4: Storing function arguments in std::tuple 302
Section 52.5: std::function with lambda and std::bind 303
Section 52.6: `function` overhead 304
Chapter 53: std::forward_list 305

Section 53.1: Example ... 305
Section 53.2: Methods ... 305
Chapter 54: std::pair ... 307
Section 54.1: Compare operators ... 307
Section 54.2: Creating a Pair and accessing the elements ... 307
Chapter 55: std::atomics ... 309
Section 55.1: atomic types ... 309
Chapter 56: std::variant ... 311
Section 56.1: Create pseudo-method pointers ... 311
Section 56.2: Basic std::variant use ... 312
Section 56.3: Constructing a `std::variant` ... 313
Chapter 57: std::iomanip ... 314
Section 57.1: std::setprecision ... 314
Section 57.2: std::setfill ... 314
Section 57.3: std::setiosflags ... 314
Section 57.4: std::setw ... 316
Chapter 58: std::any ... 317
Section 58.1: Basic usage ... 317
Chapter 59: std::set and std::multiset ... 318
Section 59.1: Changing the default sort of a set ... 318
Section 59.2: Deleting values from a set ... 320
Section 59.3: Inserting values in a set ... 321
Section 59.4: Inserting values in a multiset ... 323
Section 59.5: Searching values in set and multiset ... 323
Chapter 60: std::integer_sequence ... 325
Section 60.1: Turn a std::tuple<T...> into function parameters ... 325
Section 60.2: Create a parameter pack consisting of integers ... 326
Section 60.3: Turn a sequence of indices into copies of an element ... 326
Chapter 61: Using std::unordered_map ... 328
Section 61.1: Declaration and Usage ... 328
Section 61.2: Some Basic Functions ... 328
Chapter 62: Standard Library Algorithms ... 329
Section 62.1: std::next_permutation ... 329
Section 62.2: std::for_each ... 329
Section 62.3: std::accumulate ... 330
Section 62.4: std::find ... 331
Section 62.5: std::min_element ... 333
Section 62.6: std::find_if ... 334
Section 62.7: Using std::nth_element To Find The Median (Or Other Quantiles) ... 335
Section 62.8: std::count ... 336
Section 62.9: std::count_if ... 337
Chapter 63: The ISO C++ Standard ... 339
Section 63.1: Current Working Drafts ... 339
Section 63.2: C++17 ... 339
Section 63.3: C++11 ... 340
Section 63.4: C++14 ... 341
Section 63.5: C++98 ... 342
Section 63.6: C++03 ... 342
Section 63.7: C++20 ... 343

Chapter 64: Inline variables 344
Section 64.1: Defining a static data member in the class definition 344
Chapter 65: Random number generation 345
Section 65.1: True random value generator 345
Section 65.2: Generating a pseudo-random number 345
Section 65.3: Using the generator for multiple distributions 346
Chapter 66: Date and time using <chrono> header 347
Section 66.1: Measuring time using <chrono> 347
Section 66.2: Find number of days between two dates 347
Chapter 67: Sorting 349
Section 67.1: Sorting and sequence containers 349
Section 67.2: sorting with std::map (ascending and descending) 349
Section 67.3: Sorting sequence containers by overloaded less operator 351
Section 67.4: Sorting sequence containers using compare function 352
Section 67.5: Sorting sequence containers using lambda expressions (C++11) 353
Section 67.6: Sorting built-in arrays 354
Section 67.7: Sorting sequence containers with specifed ordering 354
Chapter 68: Enumeration 355
Section 68.1: Iteration over an enum 355
Section 68.2: Scoped enums 356
Section 68.3: Enum forward declaration in C++11 357
Section 68.4: Basic Enumeration Declaration 357
Section 68.5: Enumeration in switch statements 358
Chapter 69: Iteration 359
Section 69.1: break 359
Section 69.2: continue 359
Section 69.3: do 359
Section 69.4: while 359
Section 69.5: range-based for loop 360
Section 69.6: for 360
Chapter 70: Regular expressions 361
Section 70.1: Basic regex_match and regex_search Examples 361
Section 70.2: regex_iterator Example 361
Section 70.3: Anchors 362
Section 70.4: regex_replace Example 363
Section 70.5: regex_token_iterator Example 363
Section 70.6: Quantifiers 363
Section 70.7: Splitting a string 365
Chapter 71: Implementation-defined behavior 366
Section 71.1: Size of integral types 366
Section 71.2: Char might be unsigned or signed 368
Section 71.3: Ranges of numeric types 368
Section 71.4: Value representation of floating point types 369
Section 71.5: Overflow when converting from integer to signed integer 369
Section 71.6: Underlying type (and hence size) of an enum 370
Section 71.7: Numeric value of a pointer 370
Section 71.8: Number of bits in a byte 371
Chapter 72: Exceptions 372
Section 72.1: Catching exceptions 372

Section 72.2: Rethrow (propagate) exception ... 373
Section 72.3: Best practice: throw by value, catch by const reference ... 374
Section 72.4: Custom exception ... 375
Section 72.5: std::uncaught_exceptions ... 377
Section 72.6: Function Try Block for regular function ... 378
Section 72.7: Nested exception ... 378
Section 72.8: Function Try Blocks In constructor ... 380
Section 72.9: Function Try Blocks In destructor ... 381
Chapter 73: Lambdas ... 382
Section 73.1: What is a lambda expression? ... 382
Section 73.2: Specifying the return type ... 384
Section 73.3: Capture by value ... 385
Section 73.4: Recursive lambdas ... 386
Section 73.5: Default capture ... 388
Section 73.6: Class lambdas and capture of this ... 388
Section 73.7: Capture by reference ... 390
Section 73.8: Generic lambdas ... 390
Section 73.9: Using lambdas for inline parameter pack unpacking ... 391
Section 73.10: Generalized capture ... 393
Section 73.11: Conversion to function pointer ... 394
Section 73.12: Porting lambda functions to C++03 using functors ... 394
Chapter 74: Value Categories ... 396
Section 74.1: Value Category Meanings ... 396
Section 74.2: rvalue ... 396
Section 74.3: xvalue ... 397
Section 74.4: prvalue ... 397
Section 74.5: lvalue ... 398
Section 74.6: glvalue ... 398
Chapter 75: Preprocessor ... 399
Section 75.1: Include Guards ... 399
Section 75.2: Conditional logic and cross-platform handling ... 400
Section 75.3: X-macros ... 401
Section 75.4: Macros ... 403
Section 75.5: Predefined macros ... 406
Section 75.6: Preprocessor Operators ... 408
Section 75.7: #pragma once ... 408
Section 75.8: Preprocessor error messages ... 409
Chapter 76: Data Structures in C++ ... 410
Section 76.1: Linked List implementation in C++ ... 410
Chapter 77: Templates ... 413
Section 77.1: Basic Class Template ... 413
Section 77.2: Function Templates ... 413
Section 77.3: Variadic template data structures ... 415
Section 77.4: Argument forwarding ... 417
Section 77.5: Partial template specialization ... 418
Section 77.6: Template Specialization ... 420
Section 77.7: Alias template ... 420
Section 77.8: Explicit instantiation ... 420
Section 77.9: Non-type template parameter ... 421
Section 77.10: Declaring non-type template arguments with auto ... 422

Section 77.11: Template template parameters ... 423
Section 77.12: Default template parameter value ... 424
Chapter 78: Expression templates ... 425
Section 78.1: A basic example illustrating expression templates ... 425
Chapter 79: Curiously Recurring Template Pattern (CRTP) ... 429
Section 79.1: The Curiously Recurring Template Pattern (CRTP) ... 429
Section 79.2: CRTP to avoid code duplication ... 430
Chapter 80: Threading ... 432
Section 80.1: Creating a std::thread ... 432
Section 80.2: Passing a reference to a thread ... 434
Section 80.3: Using std::async instead of std::thread ... 434
Section 80.4: Basic Synchronization ... 435
Section 80.5: Create a simple thread pool ... 435
Section 80.6: Ensuring a thread is always joined ... 437
Section 80.7: Operations on the current thread ... 438
Section 80.8: Using Condition Variables ... 439
Section 80.9: Thread operations ... 441
Section 80.10: Thread-local storage ... 441
Section 80.11: Reassigning thread objects ... 442
Chapter 81: Thread synchronization structures ... 443
Section 81.1: std::condition_variable_any, std::cv_status ... 443
Section 81.2: std::shared_lock ... 443
Section 81.3: std::call_once, std::once_flag ... 443
Section 81.4: Object locking for efficient access ... 444
Chapter 82: The Rule of Three, Five, And Zero ... 446
Section 82.1: Rule of Zero ... 446
Section 82.2: Rule of Five ... 447
Section 82.3: Rule of Three ... 448
Section 82.4: Self-assignment Protection ... 449
Chapter 83: RAII: Resource Acquisition Is Initialization ... 451
Section 83.1: Locking ... 451
Section 83.2: ScopeSuccess (c++17) ... 452
Section 83.3: ScopeFail (c++17) ... 453
Section 83.4: Finally/ScopeExit ... 454
Chapter 84: RTTI: Run-Time Type Information ... 455
Section 84.1: dynamic_cast ... 455
Section 84.2: The typeid keyword ... 455
Section 84.3: Name of a type ... 456
Section 84.4: When to use which cast in c++ ... 456
Chapter 85: Mutexes ... 457
Section 85.1: Mutex Types ... 457
Section 85.2: std::lock ... 457
Section 85.3: std::unique_lock, std::shared_lock, std::lock_guard ... 457
Section 85.4: Strategies for lock classes: std::try_to_lock, std::adopt_lock, std::defer_lock ... 458
Section 85.5: std::mutex ... 459
Section 85.6: std::scoped_lock (C++ 17) ... 459
Chapter 86: Recursive Mutex ... 460
Section 86.1: std::recursive_mutex ... 460
Chapter 87: Semaphore ... 461

Section 87.1: Semaphore C++ 11 461
Section 87.2: Semaphore class in action 461

Chapter 88: Futures and Promises 463

Section 88.1: Async operation classes 463
Section 88.2: std::future and std::promise 463
Section 88.3: Deferred async example 463
Section 88.4: std::packaged_task and std::future 464
Section 88.5: std::future_error and std::future_errc 464
Section 88.6: std::future and std::async 465

Chapter 89: Atomic Types 468

Section 89.1: Multi-threaded Access 468

Chapter 90: Type Erasure 470

Section 90.1: A move-only `std::function` 470
Section 90.2: Erasing down to a Regular type with manual vtable 472
Section 90.3: Basic mechanism 475
Section 90.4: Erasing down to a contiguous buffer of T 476
Section 90.5: Type erasing type erasure with std::any 477

Chapter 91: Explicit type conversions 482

Section 91.1: C-style casting 482
Section 91.2: Casting away constness 482
Section 91.3: Base to derived conversion 482
Section 91.4: Conversion between pointer and integer 483
Section 91.5: Conversion by explicit constructor or explicit conversion function 484
Section 91.6: Implicit conversion 484
Section 91.7: Enum conversions 484
Section 91.8: Derived to base conversion for pointers to members 486
Section 91.9: void* to T* 486
Section 91.10: Type punning conversion 487

Chapter 92: Unnamed types 488

Section 92.1: Unnamed classes 488
Section 92.2: As a type alias 488
Section 92.3: Anonymous members 488
Section 92.4: Anonymous Union 489

Chapter 93: Type Traits 490

Section 93.1: Type Properties 490
Section 93.2: Standard type traits 491
Section 93.3: Type relations with std::is_same<T, T> 492
Section 93.4: Fundamental type traits 493

Chapter 94: Return Type Covariance 495

Section 94.1: Covariant result version of the base example, static type checking 495
Section 94.2: Covariant smart pointer result (automated cleanup) 495

Chapter 95: Layout of object types 497

Section 95.1: Class types 497
Section 95.2: Arithmetic types 499
Section 95.3: Arrays 500

Chapter 96: Type Inference 501

Section 96.1: Data Type: Auto 501
Section 96.2: Lambda auto 501
Section 96.3: Loops and auto 501

Chapter 97: Typedef and type aliases ... 503
Section 97.1: Basic typedef syntax ... 503
Section 97.2: More complex uses of typedef ... 503
Section 97.3: Declaring multiple types with typedef ... 504
Section 97.4: Alias declaration with "using" ... 504
Chapter 98: type deduction ... 505
Section 98.1: Template parameter deduction for constructors ... 505
Section 98.2: Auto Type Deduction ... 505
Section 98.3: Template Type Deduction ... 506
Chapter 99: Trailing return type ... 508
Section 99.1: Avoid qualifying a nested type name ... 508
Section 99.2: Lambda expressions ... 508
Chapter 100: Alignment ... 509
Section 100.1: Controlling alignment ... 509
Section 100.2: Querying the alignment of a type ... 509
Chapter 101: Perfect Forwarding ... 511
Section 101.1: Factory functions ... 511
Chapter 102: decltype ... 512
Section 102.1: Basic Example ... 512
Section 102.2: Another example ... 512
Chapter 103: SFINAE (Substitution Failure Is Not An Error) ... 513
Section 103.1: What is SFINAE ... 513
Section 103.2: void_t ... 513
Section 103.3: enable_if ... 515
Section 103.4: is_detected ... 516
Section 103.5: Overload resolution with a large number of options ... 518
Section 103.6: trailing decltype in function templates ... 519
Section 103.7: enable_if_all / enable_if_any ... 520
Chapter 104: Undefined Behavior ... 522
Section 104.1: Reading or writing through a null pointer ... 522
Section 104.2: Using an uninitialized local variable ... 522
Section 104.3: Accessing an out-of-bounds index ... 523
Section 104.4: Deleting a derived object via a pointer to a base class that doesn't have a virtual destructor ... 523
Section 104.5: Extending the `std` or `posix` Namespace ... 523
Section 104.6: Invalid pointer arithmetic ... 524
Section 104.7: No return statement for a function with a non-void return type ... 525
Section 104.8: Accessing a dangling reference ... 525
Section 104.9: Integer division by zero ... 526
Section 104.10: Shifting by an invalid number of positions ... 526
Section 104.11: Incorrect pairing of memory allocation and deallocation ... 526
Section 104.12: Signed Integer Overflow ... 527
Section 104.13: Multiple non-identical definitions (the One Definition Rule) ... 527
Section 104.14: Modifying a const object ... 528
Section 104.15: Returning from a [[noreturn]] function ... 529
Section 104.16: Infinite template recursion ... 529
Section 104.17: Overflow during conversion to or from floating point type ... 530
Section 104.18: Modifying a string literal ... 530
Section 104.19: Accessing an object as the wrong type ... 530

Section 104.20: Invalid derived-to-base conversion for pointers to members 531
Section 104.21: Destroying an object that has already been destroyed 531
Section 104.22: Access to nonexistent member through pointer to member 532
Section 104.23: Invalid base-to-derived static cast 532
Section 104.24: Floating point overflow 532
Section 104.25: Calling (Pure) Virtual Members From Constructor Or Destructor 532
Section 104.26: Function call through mismatched function pointer type 533

Chapter 105: Overload resolution 534

Section 105.1: Categorization of argument to parameter cost 534
Section 105.2: Arithmetic promotions and conversions 534
Section 105.3: Overloading on Forwarding Reference 535
Section 105.4: Exact match 536
Section 105.5: Overloading on constness and volatility 536
Section 105.6: Name lookup and access checking 537
Section 105.7: Overloading within a class hierarchy 538
Section 105.8: Steps of Overload Resolution 539

Chapter 106: Move Semantics 541

Section 106.1: Move semantics 541
Section 106.2: Using std::move to reduce complexity from O(n^2) to O(n) 541
Section 106.3: Move constructor 544
Section 106.4: Re-use a moved object 546
Section 106.5: Move assignment 546
Section 106.6: Using move semantics on containers 547

Chapter 107: Pimpl Idiom 549

Section 107.1: Basic Pimpl idiom 549

Chapter 108: auto 551

Section 108.1: Basic auto sample 551
Section 108.2: Generic lambda (C++14) 551
Section 108.3: auto and proxy objects 552
Section 108.4: auto and Expression Templates 552
Section 108.5: auto, const, and references 553
Section 108.6: Trailing return type 553

Chapter 109: Copy Elision 555

Section 109.1: Purpose of copy elision 555
Section 109.2: Guaranteed copy elision 556
Section 109.3: Parameter elision 557
Section 109.4: Return value elision 557
Section 109.5: Named return value elision 557
Section 109.6: Copy initialization elision 558

Chapter 110: Fold Expressions 559

Section 110.1: Unary Folds 559
Section 110.2: Binary Folds 559
Section 110.3: Folding over a comma 560

Chapter 111: Unions 561

Section 111.1: Undefined Behavior 561
Section 111.2: Basic Union Features 561
Section 111.3: Typical Use 561

Chapter 112: Design pattern implementation in C++ 563

Section 112.1: Adapter Pattern 563

Section 112.2: Observer pattern 565
Section 112.3: Factory Pattern 568
Section 112.4: Builder Pattern with Fluent API 568
Chapter 113: Singleton Design Pattern 572
Section 113.1: Lazy Initialization 572
Section 113.2: Static deinitialization-safe singleton 573
Section 113.3: Thread-safe Singeton 573
Section 113.4: Subclasses 573
Chapter 114: User-Defined Literals 575
Section 114.1: Self-made user-defined literal for binary 575
Section 114.2: Standard user-defined literals for duration 575
Section 114.3: User-defined literals with long double values 576
Section 114.4: Standard user-defined literals for strings 576
Section 114.5: Standard user-defined literals for complex 577
Chapter 115: Memory management 578
Section 115.1: Free Storage (Heap, Dynamic Allocation ...) 578
Section 115.2: Placement new 579
Section 115.3: Stack 580
Chapter 116: C++11 Memory Model 581
Section 116.1: Need for Memory Model 582
Section 116.2: Fence example 584
Chapter 117: Scopes 585
Section 117.1: Global variables 585
Section 117.2: Simple block scope 585
Chapter 118: static_assert 587
Section 118.1: static_assert 587
Chapter 119: constexpr 588
Section 119.1: constexpr variables 588
Section 119.2: Static if statement 589
Section 119.3: constexpr functions 590
Chapter 120: One Definition Rule (ODR) 592
Section 120.1: ODR violation via overload resolution 592
Section 120.2: Multiply defined function 592
Section 120.3: Inline functions 593
Chapter 121: Unspecified behavior 595
Section 121.1: Value of an out-of-range enum 595
Section 121.2: Evaluation order of function arguments 595
Section 121.3: Result of some reinterpret_cast conversions 596
Section 121.4: Space occupied by a reference 597
Section 121.5: Moved-from state of most standard library classes 597
Section 121.6: Result of some pointer comparisons 598
Section 121.7: Static cast from bogus void* value 598
Section 121.8: Order of initialization of globals across TU 598
Chapter 122: Argument Dependent Name Lookup 600
Section 122.1: What functions are found 600
Chapter 123: Attributes 601
Section 123.1: [[fallthrough]] 601
Section 123.2: [[nodiscard]] 601
Section 123.3: [[deprecated]] and [[deprecated("reason")]] 602

Section 123.4: [[maybe_unused]] ... 602
Section 123.5: [[noreturn]] ... 603
Chapter 124: Recursion in C++ ... 605
Section 124.1: Using tail recursion and Fibonnaci-style recursion to solve the Fibonnaci sequence ... 605
Section 124.2: Recursion with memoization ... 605
Chapter 125: Arithmitic Metaprogramming ... 607
Section 125.1: Calculating power in O(log n) ... 607
Chapter 126: Callable Objects ... 609
Section 126.1: Function Pointers ... 609
Section 126.2: Classes with operator() (Functors) ... 609
Chapter 127: Client server examples ... 611
Section 127.1: Hello TCP Client ... 611
Section 127.2: Hello TCP Server ... 612
Chapter 128: Const Correctness ... 616
Section 128.1: The Basics ... 616
Section 128.2: Const Correct Class Design ... 616
Section 128.3: Const Correct Function Parameters ... 618
Section 128.4: Const Correctness as Documentation ... 620
Chapter 129: Parameter packs ... 624
Section 129.1: A template with a parameter pack ... 624
Section 129.2: Expansion of a parameter pack ... 624
Chapter 130: Build Systems ... 625
Section 130.1: Generating Build Environment with CMake ... 625
Section 130.2: Compiling with GNU make ... 626
Section 130.3: Building with SCons ... 628
Section 130.4: Autotools (GNU) ... 628
Section 130.5: Ninja ... 629
Section 130.6: NMAKE (Microsoft Program Maintenance Utility) ... 629
Chapter 131: Concurrency With OpenMP ... 630
Section 131.1: OpenMP: Parallel Sections ... 630
Section 131.2: OpenMP: Parallel Sections ... 630
Section 131.3: OpenMP: Parallel For Loop ... 631
Section 131.4: OpenMP: Parallel Gathering / Reduction ... 631
Chapter 132: Resource Management ... 633
Section 132.1: Resource Acquisition Is Initialization ... 633
Section 132.2: Mutexes & Thread Safety ... 634
Chapter 133: Storage class specifiers ... 636
Section 133.1: extern ... 636
Section 133.2: register ... 637
Section 133.3: static ... 637
Section 133.4: auto ... 638
Section 133.5: mutable ... 638
Chapter 134: Linkage specifications ... 640
Section 134.1: Signal handler for Unix-like operating system ... 640
Section 134.2: Making a C library header compatible with C++ ... 640
Chapter 135: Digit separators ... 642
Section 135.1: Digit Separator ... 642
Chapter 136: C incompatibilities ... 643

Section 136.1: Reserved Keywords 643
Section 136.2: Weakly typed pointers 643
Section 136.3: goto or switch 643
Chapter 137: Side by Side Comparisons of classic C++ examples solved via C++ vs C++11 vs C++14 vs C++17 644
Section 137.1: Looping through a container 644
Chapter 138: Compiling and Building 645
Section 138.1: Compiling with GCC 645
Section 138.2: Compiling with Visual Studio (Graphical Interface) - Hello World 646
Section 138.3: Online Compilers 651
Section 138.4: Compiling with Visual C++ (Command Line) 653
Section 138.5: Compiling with Clang 656
Section 138.6: The C++ compilation process 656
Section 138.7: Compiling with Code::Blocks (Graphical interface) 658
Chapter 139: Common compile/linker errors (GCC) 661
Section 139.1: undefined reference to `***' 661
Section 139.2: error: '***' was not declared in this scope 661
Section 139.3: fatal error: ***: No such file or directory 663
Chapter 140: More undefined behaviors in C++ 664
Section 140.1: Referring to non-static members in initializer lists 664
Chapter 141: Unit Testing in C++ 665
Section 141.1: Google Test 665
Section 141.2: Catch 665
Chapter 142: C++ Debugging and Debug-prevention Tools & Techniques 667
Section 142.1: Static analysis 667
Section 142.2: Segfault analysis with GDB 668
Section 142.3: Clean code 669
Chapter 143: Optimization in C++ 671
Section 143.1: Introduction to performance 671
Section 143.2: Empty Base Class Optimization 671
Section 143.3: Optimizing by executing less code 672
Section 143.4: Using efficient containers 673
Section 143.5: Small Object Optimization 674
Chapter 144: Optimization 676
Section 144.1: Inline Expansion/Inlining 676
Section 144.2: Empty base optimization 676
Chapter 145: Profiling 678
Section 145.1: Profiling with gcc and gprof 678
Section 145.2: Generating callgraph diagrams with gperf2dot 678
Section 145.3: Profiling CPU Usage with gcc and Google Perf Tools 679
Chapter 146: Refactoring Techniques 681
Section 146.1: Goto Cleanup 681
Credits 682
You may also like 690

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/CPlusPlusBook

This is an unofficial free book created for educational purposes and is not affiliated with official C++ group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with C++

Version	Standard	Release Date
C++98	ISO/IEC 14882:1998	1998-09-01
C++03	ISO/IEC 14882:2003	2003-10-16
C++11	ISO/IEC 14882:2011	2011-09-01
C++14	ISO/IEC 14882:2014	2014-12-15
C++17	TBD	2017-01-01
C++20	TBD	2020-01-01

Section 1.1: Hello World

This program prints `Hello World!` to the standard output stream:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

See it live on Coliru.

Analysis

Let's examine each part of this code in detail:

- `#include <iostream>` is a **preprocessor directive** that includes the content of the standard C++ header file `iostream`.

  `iostream` is a **standard library header file** that contains definitions of the standard input and output streams. These definitions are included in the `std` namespace, explained below.

  The **standard input/output (I/O) streams** provide ways for programs to get input from and output to an external system -- usually the terminal.

- `int main() { ... }` defines a new function named `main`. By convention, the `main` function is called upon execution of the program. There must be only one `main` function in a C++ program, and it must always return a number of the `int` type.

  Here, the `int` is what is called the function's return type. The value returned by the `main` function is an **exit code.**

  By convention, a program exit code of `0` or `EXIT_SUCCESS` is interpreted as success by a system that executes the program. Any other return code is associated with an error.

  If no `return` statement is present, the `main` function (and thus, the program itself) returns `0` by default. In this example, we don't need to explicitly write `return 0;`.

  All other functions, except those that return the `void` type, must explicitly return a value according to their return type, or else must not return at all.

- `std::cout << "Hello World!" << std::endl;` prints "Hello World!" to the standard output stream:
  - `std` is a namespace, and `::` is the **scope resolution operator** that allows look-ups for objects by name within a namespace.

    There are many namespaces. Here, we use `::` to show we want to use `cout` from the `std` namespace. For more information refer to Scope Resolution Operator - Microsoft Documentation.
  - `std::cout` is the **standard output stream** object, defined in `iostream`, and it prints to the standard output (`stdout`).
  - `<<` is, *in this context*, the **stream insertion operator**, so called because it *inserts* an object into the *stream* object.

    The standard library defines the `<<` operator to perform data insertion for certain data types into output streams. `stream << content` inserts `content` into the stream and returns the same, but updated stream. This allows stream insertions to be chained: `std::cout << "Foo" << " Bar";` prints "FooBar" to the console.
  - `"Hello World!"` is a **character string literal**, or a "text literal." The stream insertion operator for character string literals is defined in file `iostream`.
  - `std::endl` is a special **I/O stream manipulator** object, also defined in file `iostream`. Inserting a manipulator into a stream changes the state of the stream.

    The stream manipulator `std::endl` does two things: first it inserts the end-of-line character and then it flushes the stream buffer to force the text to show up on the console. This ensures that the data inserted into the stream actually appear on your console. (Stream data is usually stored in a buffer and then "flushed" in batches unless you force a flush immediately.)

    An alternate method that avoids the flush is:

    ```
    std::cout << "Hello World!\n";
    ```

    where `\n` is the **character escape sequence** for the newline character.
  - The semicolon (`;`) notifies the compiler that a statement has ended. All C++ statements and class definitions require an ending/terminating semicolon.

Section 1.2: Comments

A **comment** is a way to put arbitrary text inside source code without having the C++ compiler interpret it with any functional meaning. Comments are used to give insight into the design or method of a program.

There are two types of comments in C++:

Single-Line Comments

The double forward-slash sequence `//` will mark all text until a newline as a comment:

```
int main()
{
```

```
    // This is a single-line comment.
    int a;  // this also is a single-line comment
    int i;  // this is another single-line comment
}
```

C-Style/Block Comments

The sequence `/*` is used to declare the start of the comment block and the sequence `*/` is used to declare the end of comment. All text between the start and end sequences is interpreted as a comment, even if the text is otherwise valid C++ syntax. These are sometimes called "C-style" comments, as this comment syntax is inherited from C++'s predecessor language, C:

```
int main()
{
    /*
     *  This is a block comment.
     */
    int a;
}
```

In any block comment, you can write anything you want. When the compiler encounters the symbol `*/`, it terminates the block comment:

```
int main()
{
    /* A block comment with the symbol /*
       Note that the compiler is not affected by the second /*
       however, once the end-block-comment symbol is reached,
       the comment ends.
    */
    int a;
}
```

The above example is valid C++ (and C) code. However, having additional `/*` inside a block comment might result in a warning on some compilers.

Block comments can also start and end *within* a single line. For example:

```
void SomeFunction(/* argument 1 */ int a, /* argument 2 */ int b);
```

Importance of Comments

As with all programming languages, comments provide several benefits:

- Explicit documentation of code to make it easier to read/maintain
- Explanation of the purpose and functionality of code
- Details on the history or reasoning behind the code
- Placement of copyright/licenses, project notes, special thanks, contributor credits, etc. directly in the source code.

However, comments also have their downsides:

- They must be maintained to reflect any changes in the code
- Excessive comments tend to make the code *less* readable

The need for comments can be reduced by writing clear, self-documenting code. A simple example is the use of explanatory names for variables, functions, and types. Factoring out logically related tasks into discrete functions goes hand-in-hand with this.

Comment markers used to disable code

During development, comments can also be used to quickly disable portions of code without deleting it. This is often useful for testing or debugging purposes, but is not good style for anything other than temporary edits. This is often referred to as “commenting out”.

Similarly, keeping old versions of a piece of code in a comment for reference purposes is frowned upon, as it clutters files while offering little value compared to exploring the code's history via a versioning system.

Section 1.3: The standard C++ compilation process

Executable C++ program code is usually produced by a compiler.

A **compiler** is a program that translates code from a programming language into another form which is (more) directly executable for a computer. Using a compiler to translate code is called **compilation.**

C++ inherits the form of its compilation process from its "parent" language, C. Below is a list showing the four major steps of compilation in C++:

1. The C++ preprocessor copies the contents of any included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into assembly language appropriate for the platform.
3. The assembler code generated by the compiler is assembled into appropriate object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.

- Note: some compiled code is linked together, but not to create a final program. Usually, this "linked" code can also be packaged into a format that can be used by other programs. This "bundle of packaged, usable code" is what C++ programmers refer to as a **library.**

Many C++ compilers may also merge or un-merge certain parts of the compilation process for ease or for additional analysis. Many C++ programmers will use different tools, but all of the tools will generally follow this generalized process when they are involved in the production of a program.

The link below extends this discussion and provides a nice graphic to help. [1]:
http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Section 1.4: Function

A **function** is a unit of code that represents a sequence of statements.

Functions can accept **arguments** or values and **return** a single value (or not). To use a function, a **function call** is used on argument values and the use of the function call itself is replaced with its return value.

Every function has a **type signature** -- the types of its arguments and the type of its return type.

Functions are inspired by the concepts of the procedure and the mathematical function.

- Note: C++ functions are essentially procedures and do not follow the exact definition or rules of mathematical functions.

Functions are often meant to perform a specific task. and can be called from other parts of a program. A function must be declared and defined before it is called elsewhere in a program.

- Note: popular function definitions may be hidden in other included files (often for convenience and reuse across many files). This is a common use of header files.

Function Declaration

A **function declaration** is declares the existence of a function with its name and type signature to the compiler. The syntax is as the following:

```
int add2(int i); // The function is of the type (int) -> (int)
```

In the example above, the `int add2(int i)` function declares the following to the compiler:

- The **return type** is `int`.
- The **name** of the function is `add2`.
- The **number of arguments** to the function is 1:
    - The first argument is of the type `int`.
    - The first argument will be referred to in the function's contents by the name `i`.

The argument name is optional; the declaration for the function could also be the following:

```
int add2(int); // Omitting the function arguments' name is also permitted.
```

Per the **one-definition rule**, a function with a certain type signature can only be declared or defined once in an entire C++ code base visible to the C++ compiler. In other words, functions with a specific type signature cannot be re-defined -- they must only be defined once. Thus, the following is not valid C++:

```
int add2(int i);  // The compiler will note that add2 is a function (int) -> int
int add2(int j);  // As add2 already has a definition of (int) -> int, the compiler
                  // will regard this as an error.
```

If a function returns nothing, its return type is written as `void`. If it takes no parameters, the parameter list should be empty.

```
void do_something(); // The function takes no parameters, and does not return anything.
                     // Note that it can still affect variables it has access to.
```

Function Call

A function can be called after it has been declared. For example, the following program calls `add2` with the value of `2` within the function of `main`:

```
#include <iostream>

int add2(int i);    // Declaration of add2

// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.

int main()
{
    std::cout << add2(2) << "\n";  // add2(2) will be evaluated at this point,
                                   // and the result is printed.
    return 0;
}
```

Here, `add2(2)` is the syntax for a function call.

Function Definition

A *function definition** is similar to a declaration, except it also contains the code that is executed when the function is called within its body.

An example of a function definition for `add2` might be:

```
int add2(int i)       // Data that is passed into (int i) will be referred to by the name i
{                     // while in the function's curly brackets or "scope."

    int j = i + 2;    // Definition of a variable j as the value of i+2.
    return j;         // Returning or, in essence, substitution of j for a function call to
                      // add2.
}
```

Function Overloading

You can create multiple functions with the same name but different parameters.

```
int add2(int i)           // Code contained in this definition will be evaluated
{                         // when add2() is called with one parameter.
    int j = i + 2;
    return j;
}

int add2(int i, int j)    // However, when add2() is called with two parameters, the
{                         // code from the initial declaration will be overloaded,
    int k = i + j + 2 ;   // and the code in this declaration will be evaluated
    return k;             // instead.
}
```

Both functions are called by the same name `add2`, but the actual function that is called depends directly on the amount and type of the parameters in the call. In most cases, the C++ compiler can compute which function to call. In some cases, the type must be explicitly stated.

Default Parameters

Default values for function parameters can only be specified in function declarations.

```
int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
    return a * b;               // If multiply() is called with one parameter, the
}                               // value will be multiplied by the default, 7.
```

In this example, `multiply()` can be called with one or two parameters. If only one parameter is given, `b` will have default value of 7. Default arguments must be placed in the latter arguments of the function. For example:

```
int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);      // This is illegal since int a is in the former
```

Special Function Calls - Operators

There exist special function calls in C++ which have different syntax than `name_of_function(value1, value2, value3)`. The most common example is that of operators.

Certain special character sequences that will be reduced to function calls by the compiler, such as `!`, `+`, `-`, `*`, `%`, and `<<` and many more. These special characters are normally associated with non-programming usage or are used for aesthetics (e.g. the + character is commonly recognized as the addition symbol both within C++ programming as well as in elementary math).

C++ handles these character sequences with a special syntax; but, in essence, each occurrence of an operator is reduced to a function call. For example, the following C++ expression:

```
3+3
```

is equivalent to the following function call:

```
operator+(3, 3)
```

All operator function names start with `operator`.

While in C++'s immediate predecessor, C, operator function names cannot be assigned different meanings by providing additional definitions with different type signatures, in C++, this is valid. "Hiding" additional function definitions under one unique function name is referred to as **operator overloading** in C++, and is a relatively common, but not universal, convention in C++.

Section 1.5: Visibility of function prototypes and declarations

In C++, code must be declared or defined before usage. For example, the following produces a compile time error:

```
int main()
{
  foo(2); // error: foo is called, but has not yet been declared
}

void foo(int x) // this later definition is not known in main
{
}
```

There are two ways to resolve this: putting either the definition or declaration of `foo()` before its usage in `main()`. Here is one example:

```
void foo(int x) {}  //Declare the foo function and body first

int main()
{
  foo(2); // OK: foo is completely defined beforehand, so it can be called here.
}
```

However it is also possible to "forward-declare" the function by putting only a "prototype" declaration before its usage and then defining the function body later:

```
void foo(int);  // Prototype declaration of foo, seen by main
                // Must specify return type, name, and argument list types
int main()
{
  foo(2); // OK: foo is known, called even though its body is not yet defined
}

void foo(int x) //Must match the prototype
{
    // Define body of foo here
}
```

The prototype must specify the return type (`void`), the name of the function (`foo`), and the argument list variable types (`int`), but the names of the arguments are NOT required.

One common way to integrate this into the organization of source files is to make a header file containing all of the prototype declarations:

```
// foo.h
void foo(int); // prototype declaration
```

and then provide the full definition elsewhere:

```
// foo.cpp --> foo.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
void foo(int x) { } // foo's body definition
```

and then, once compiled, link the corresponding object file `foo.o` into the compiled object file where it is used in the linking phase, `main.o`:

```
// main.cpp --> main.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
int main() { foo(2); } // foo is valid to call because its prototype declaration was beforehand.
// the prototype and body definitions of foo are linked through the object files
```

An “unresolved external symbol” error occurs when the function *prototype* and *call* exist, but the function *body* is not defined. These can be trickier to resolve as the compiler won't report the error until the final linking stage, and it doesn't know which line to jump to in the code to show the error.

Section 1.6: Preprocessor

The preprocessor is an important part of the compiler.

It edits the source code, cutting some bits out, changing others, and adding other things.

In source files, we can include preprocessor directives. These directives tells the preprocessor to perform specific actions. A directive starts with a # on a new line. Example:

```
#define ZERO 0
```

The first preprocessor directive you will meet is probably the

```
#include <something>
```

directive. What it does is takes all of `something` and inserts it in your file where the directive was. The hello world program starts with the line

```
#include <iostream>
```

This line adds the functions and objects that let you use the standard input and output.

The C language, which also uses the preprocessor, does not have as many header files as the C++ language, but in C++ you can use all the C header files.

The next important directive is probably the

```
#define something something_else
```

directive. This tells the preprocessor that as it goes along the file, it should replace every occurrence of `something` with `something_else`. It can also make things similar to functions, but that probably counts as advanced C++.

The `something_else` is not needed, but if you define `something` as nothing, then outside preprocessor directives, all occurrences of `something` will vanish.

This actually is useful, because of the `#if`,`#else` and `#ifdef` directives. The format for these would be the following:

```
#if something==true
//code
#else
//more code
#endif

#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

These directives insert the code that is in the true bit, and deletes the false bits. this can be used to have bits of code that are only included on certain operating systems, without having to rewrite the whole code.

Chapter 2: Literals

Traditionally, a literal is an expression denoting a constant whose type and value are evident from its spelling. For example, `42` is a literal, while `x` is not since one must see its declaration to know its type and read previous lines of code to know its value.

However, C++11 also added user-defined literals, which are not literals in the traditional sense but can be used as a shorthand for function calls.

Section 2.1: this

Within a member function of a class, the keyword `this` is a pointer to the instance of the class on which the function was called. `this` cannot be used in a static member function.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

The type of `this` depends on the cv-qualification of the member function: if `X::f` is `const`, then the type of `this` within `f` is `const X*`, so `this` cannot be used to modify non-static data members from within a `const` member function. Likewise, `this` inherits `volatile` qualification from the function it appears in.

`Version ≥ C++11`

`this` can also be used in a *brace-or-equal-initializer* for a non-static data member.

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

`this` is an rvalue, so it cannot be assigned to.

Section 2.2: Integer literal

An integer literal is a primary expression of the form

- decimal-literal

It is a non-zero decimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9), followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- octal-literal

It is the digit zero (0) followed by zero or more octal digits (0, 1, 2, 3, 4, 5, 6, 7) `int o = 052`

- hex-literal

It is the character sequence 0x or the character sequence 0X followed by one or more hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F)

`int x = 0x2a; int X = 0X2A;`

- binary-literal (since C++14)

It is the character sequence 0b or the character sequence 0B followed by one or more binary digits (0, 1)

`int b = 0b101010; // C++14`

Integer-suffix, if provided, may contain one or both of the following (if both are provided, they may appear in any order:

- unsigned-suffix (the character u or the character U)

`unsigned int u_1 = 42u;`

- long-suffix (the character l or the character L) or the long-long-suffix (the character sequence ll or the character sequence LL) (since C++11)

The following variables are also initialized to the same value:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Notes

Letters in the integer literals are case-insensitive: 0xDeAdBaBeU and 0XdeadBABEu represent the same number (one exception is the long-long-suffix, which is either ll or LL, never lL or Ll)

There are no negative integer literals. Expressions such as -1 apply the unary minus operator to the value represented by the literal, which may involve implicit type conversions.

In C prior to C99 (but not in C++), unsuffixed decimal values that do not fit in long int are allowed to have the type unsigned long int.

When used in a controlling expression of #if or #elif, all signed integer constants act as if they have type std::intmax_t and all unsigned integer constants act as if they have type std::uintmax_t.

Section 2.3: true

A keyword denoting one of the two possible values of type `bool`.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

Section 2.4: false

A keyword denoting one of the two possible values of type bool.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

Section 2.5: nullptr

Version ≥ C++11

A keyword denoting a null pointer constant. It can be converted to any pointer or pointer-to-member type, yielding a null pointer of the resulting type.

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

Note that nullptr is not itself a pointer. The type of nullptr is a fundamental type known as std::nullptr_t.

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

Chapter 3: operator precedence

Section 3.1: Logical && and || operators: short-circuit

&& has precedence over ||, this means that parentheses are placed to evaluate what would be evaluated together.

c++ uses short-circuit evaluation in && and || to not do unnecessary executions.
If the left hand side of || returns true the right hand side does not need to be evaluated anymore.

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

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}


int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
                // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //"Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||,
    //B being false we do not have to evaluate C to know that the result is false



    result =
        True("A") || False("B") && False("C");
                // eq. True("A") || (False("B") && False("C"))
    cout << result << " :=====================" << endl;
    //TrueA
    //"Short-circuit evaluation skip of B"
    //"Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //        the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //"Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
}
```

Section 3.2: Unary Operators

Unary operators act on the object upon which they are called and have high precedence. (See Remarks)

When used postfix, the action occurs only after the entire operation is evaluated, leading to some interesting arithmetics:

```
int a = 1;
++a;            // result: 2
a--;            // result: 1
int minusa=-a;  // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2;      // equal to: (a==4) 4 / 2   result: 2 ('a' incremented postfix)
cout << a << endl;  // prints 5!
int d = ++a/2;      // equal to: (a+1) == 6 / 2 result: 3

int arr[4] =  {1,2,3,4};

int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;     // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl;  // prints  2

int e = arr[0]++;       // receives the value of arr[0] before it is incremented
std::cout << e << std::endl;      // prints 1
std::cout << *ptr2 << std::endl;  // prints arr[0] which is now 2
```

Section 3.3: Arithmetic operators

Arithmetic operators in C++ have the same precedence as they do in mathematics:

Multiplication and division have left associativity(meaning that they will be evaluated from left to right) and they have higher precedence than addition and subtraction, which also have left associativity.

We can also force the precedence of expression using parentheses `( )`. Just the same way as you would do that in normal mathematics.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;          // equal to: 2+(4/2)         result: 4
int b = (3+3)/2;        // equal to: (3+3)/2         result: 3

//With Multiplication

int c = 3+4/2*6;        // equal to: 3+((4/2)*6)     result: 15
int d = 3*(3+6)/9;      // equal to: (3*(3+6))/9     result: 3

//Division and Modulo

int g = 3-3%1;          // equal to: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);        // equal to: 3 % 1 = 0  3 - 0 = 3
int i = 3-3/1%3;        // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int l = 3-(3/1)%3;      // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int m = 3-(3/(1%3));    // equal to: 1 % 3 = 1  3 / 1 = 3  3 - 3 = 0
```

Section 3.4: Logical AND and OR operators

These operators have the usual precedence in C++: AND before OR.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

This code is equivalent to the following:

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

Adding the parenthesis does not change the behavior, though, it does make it easier to read. By adding these parentheses, no confusion exist about the intent of the writer.

Chapter 4: Floating Point Arithmetic

Section 4.1: Floating Point Numbers are Weird

The first mistake that nearly every single programmer makes is presuming that this code will work as intended:

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
    total += a;
}
```

The novice programmer assumes that this will sum up every single number in the range `0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99`, to yield the result `199`—the mathematically correct answer.

Two things happen that make this untrue:

1. The program as written never concludes. `a` never becomes equal to `2`, and the loop never terminates.
2. If we rewrite the loop logic to check `a < 2` instead, the loop terminates, but the total ends up being something different from `199`. On IEEE754-compliant machines, it will often sum up to about `201` instead.

The reason that this happens is that **Floating Point Numbers represent Approximations of their assigned values**.

The classical example is the following computation:

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //This never prints on IEEE754-compliant machines
    std::cout << "This Computer is Magic!" << std::endl;
else
    std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Though what we the programmer see is three numbers written in base10, what the compiler (and the underlying hardware) see are binary numbers. Because `0.1`, `0.2`, and `0.3` require perfect division by `10`—which is quite easy in a base-10 system, but impossible in a base-2 system—these numbers have to be stored in imprecise formats, similar to how the number `1/3` has to be stored in the imprecise form `0.333333333333333...` in base-10.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a =     0011111110111001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.1
double b =     0011111111001001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.2
double c =     0011111111010011001100110011001100110011001100110011001100110011; //imperfect
representation of 0.3
double a + b = 0011111111010011001100110011001100110011001100110011001100110100; //Note that this
is not quite equal to the "canonical" 0.3!
```

Chapter 5: Bit Operators

Section 5.1: | - bitwise OR

```
int a = 5;     // 0101b  (0x05)
int b = 12;    // 1100b  (0x0C)
int c = a | b; // 1101b  (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 5, b = 12, c = 13
```

Why

A bit wise `OR` operates on the bit level and uses the following Boolean truth table:

```
true OR true = true
true OR false = true
false OR false = false
```

When the binary value for `a` (`0101`) and the binary value for `b` (`1100`) are `OR`'ed together we get the binary value of `1101`:

```
int a = 0 1 0 1
int b = 1 1 0 0 |
        ---------
int c = 1 1 0 1
```

The bit wise OR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `|=`:

```
int a = 5;  // 0101b  (0x05)
a |= 12;    // a = 0101b | 1101b
```

Section 5.2: ^ - bitwise XOR (exclusive OR)

```
int a = 5;     // 0101b  (0x05)
int b = 9;     // 1001b  (0x09)
int c = a ^ b; // 1100b  (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

```
a = 5, b = 9, c = 12
```

Why

A bit wise `XOR` (exclusive or) operates on the bit level and uses the following Boolean truth table:

```
true OR true = false
true OR false = true
false OR false = false
```

Notice that with an XOR operation `true OR true = false` where as with operations `true AND/OR true = true`, hence the exclusive nature of the XOR operation.

Using this, when the binary value for `a` (`0101`) and the binary value for `b` (`1001`) are `XOR`'ed together we get the binary value of `1100`:

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
        ---------
int c = 1 1 0 0
```

The bit wise XOR does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `^=`:

```
int a = 5;  // 0101b  (0x05)
a ^= 9;    // a = 0101b ^ 1001b
```

The bit wise XOR can be utilized in many ways and is often utilized in bit mask operations for encryption and compression.

Note: The following example is often shown as an example of a nice trick. But should not be used in production code (there are better ways `std::swap()` to achieve the same result).

You can also utilize an XOR operation to swap two variables without a temporary:

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

To productionalize this you need to add a check to make sure it can be used.

```
void doXORSwap(int& a, int& b)
{
    // Need to add a check to make sure you are not swapping the same
    // variable with itself. Otherwise it will zero the value.
    if (&a != &b)
    {
        // XOR swap
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

So though it looks like a nice trick in isolation it is not useful in real code. xor is not a base logical operation,but a combination of others: a^c=~(a&c)&(a|c)

also in 2015+ compilers variables may be assigned as binary:

```
int cn=0b0111;
```

Section 5.3: & - bitwise AND

```
int a = 6;     // 0110b  (0x06)
int b = 10;    // 1010b  (0x0A)
int c = a & b; // 0010b  (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Output

`a = 6, b = 10, c = 2`

Why

A bit wise `AND` operates on the bit level and uses the following Boolean truth table:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

When the binary value for `a` (`0110`) and the binary value for `b` (`1010`) are `AND`'ed together we get the binary value of `0010`:

```
int a = 0 1 1 0
int b = 1 0 1 0 &
        ---------
int c = 0 0 1 0
```

The bit wise AND does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `&=`:

```
int a = 5;  // 0101b  (0x05)
a &= 10;    // a = 0101b & 1010b
```

Section 5.4: << - left shift

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Output

`a = 1, b = 2`

Why

The left bit wise shift will shift the bits of the left hand value (`a`) the number specified on the right (`1`), essentially padding the least significant bits with 0's, so shifting the value of `5` (binary `0000 0101`) to the left 4 times (e.g. `5 << 4`) will yield the value of `80` (binary `0101 0000`). You might note that shifting a value to the left 1 time is also the same as multiplying the value by 2, example:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
}

a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a *= 2;
}
```

But it should be noted that the left shift operation will shift *all* bits to the left, including the sign bit, example:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;     // 1111 1111 1111 1111 1111 1111 1111 1110

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Possible output: `a = 2147483647, b = -2`

While some compilers will yield results that seem expected, it should be noted that if you left shift a signed number so that the sign bit is affected, the result is **undefined**. It is also **undefined** if the number of bits you wish to shift by is a negative number or is larger than the number of bits the type on the left can hold, example:

```
int a = 1;
int b = a << -1;  // undefined behavior
char c = a << 20; // undefined behavior
```

The bit wise left shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `<<=`:

```
int a = 5;  // 0101b
a <<= 1;    // a = a << 1;
```

Section 5.5: >> - right shift

```
int a = 2;      // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Output

```
a = 2, b = 1
```

Why

The right bit wise shift will shift the bits of the left hand value (`a`) the number specified on the right (`1`); it should be noted that while the operation of a right shift is standard, what happens to the bits of a right shift on a *signed negative* number is *implementation defined* and thus cannot be guaranteed to be portable, example:

```
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
```

It is also undefined if the number of bits you wish to shift by is a negative number, example:

```
int a = 1;
int b = a >> -1;  // undefined behavior
```

The bit wise right shift does not change the value of the original values unless specifically assigned to using the bit wise assignment compound operator `>>=`:

```
int a = 2;  // 0010b
a >>= 1;    // a = a >> 1;
```

Chapter 6: Bit Manipulation

Section 6.1: Remove rightmost set bit

C-style bit-manipulation

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
unsigned"); // For c++11 and later
    return n & (n - 1);
}
```

Explanation

- if `n` is zero, we have `0 & 0xFF..FF` which is zero
- else `n` can be written `0bxxxxxx10..00` and `n - 1` is `0bxxxxxx011..11`, so `n & (n - 1)` is `0bxxxxxx000..00`.

Section 6.2: Set all bits

C-style bit-manipulation

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(See here for an explanation of why this works and is actually the best approach.)

Using std::bitset

```
std::bitset<10> x;
x.set(); // Sets all bits to '1'
```

Section 6.3: Toggling a bit

C-style bit-manipulation

A bit can be toggled using the XOR operator (`^`).

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

Using std::bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip();  // num is now 1110 (flips all bits)
```

Section 6.4: Checking a bit

C-style bit-manipulation

The value of the bit can be obtained by shifting the number to the right `x` times and then performing bitwise AND (`&`) on it:

```
(number >> x) & 1LL;  // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

The right-shift operation may be implemented as either an arithmetic (signed) shift or a logical (unsigned) shift. If number in the expression number >> x has a signed type and a negative value, the resulting value is implementation-defined.

If we need the value of that bit directly in-place, we could instead left shift the mask:

```
(number & (1LL << x));  // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

Either can be used as a conditional, since all non-zero values are considered true.

Using std::bitset

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1);  // bit_val value is set to true;
```

Section 6.5: Counting bits set

The population count of a bitstring is often needed in cryptography and other applications and the problem has been widely studied.

The naive way requires one iteration per bit:

```
unsigned value = 1234;
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (bits = 0; value; value >>= 1)
  bits += value & 1;
```

A nice trick (based on Remove rightmost set bit) is:

```
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (; value; ++bits)
  value &= value - 1;
```

It goes through as many iterations as there are set bits, so it's good when value is expected to have few nonzero bits.

The method was first proposed by Peter Wegner (in CACM 3 / 322 - 1960) and it's well known since it appears in *C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

This requires 12 arithmetic operations, one of which is a multication:

```
unsigned popcount(std::uint64_t x)
{
  const std::uint64_t m1  = 0x5555555555555555;  // binary: 0101...
  const std::uint64_t m2  = 0x3333333333333333;  // binary: 00110011..
  const std::uint64_t m4  = 0x0f0f0f0f0f0f0f0f;  // binary: 0000111100001111

  x -= (x >> 1) & m1;             // put count of each 2 bits into those 2 bits
  x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
  x = (x + (x >> 4)) & m4;        // put count of each 8 bits into those 8 bits
  return (x * h01) >> 56;  // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

This kind of implementation has the best worst-case behavior (see Hamming weight for further details).

Many CPUs have a specific instruction (like x86's popcnt) and the compiler could offer a specific (**non standard**) built in function. E.g. with g++ there is:

```
int __builtin_popcount (unsigned x);
```

Section 6.6: Check if an integer is a power of 2

The `n & (n - 1)` trick (see Remove rightmost set bit) is also useful to determine if an integer is a power of 2:

```
bool power_of_2 = n && !(n & (n - 1));
```

Note that without the first part of the check (`n &&`), `0` is incorrectly considered a power of 2.

Section 6.7: Setting a bit

C-style bit manipulation

A bit can be set using the bitwise OR operator (`|`).

```
// Bit x will be set
number |= 1LL << x;
```

Using std::bitset

`set(x)` or `set(x,true)` - sets bit at position `x` to `1`.

```
std::bitset<5> num(std::string("01100"));
num.set(0);      // num is now 01101
num.set(2);      // num is still 01101
num.set(4,true); // num is now 11110
```

Section 6.8: Clearing a bit

C-style bit-manipulation

A bit can be cleared using the bitwise AND operator (`&`).

```
// Bit x will be cleared
number &= ~(1LL << x);
```

Using std::bitset

`reset(x)` or `set(x,false)` - clears the bit at position `x`.

```
std::bitset<5> num(std::string("01100"));
num.reset(2);     // num is now 01000
num.reset(0);     // num is still 01000
num.set(3,false); // num is now 00000
```

Section 6.9: Changing the nth bit to x

C-style bit-manipulation

```
// Bit n will be set if x is 1 and cleared if x is 0.
number ^= (-x ^ number) & (1LL << n);
```

Using std::bitset

`set(n,val)` - sets bit `n` to the value `val`.

```
std::bitset<5> num(std::string("00100"));
num.set(0,true);  // num is now 00101
num.set(2,false); // num is now 00001
```

Section 6.10: Bit Manipulation Application: Small to Capital Letter

One of several applications of bit manipulation is converting a letter from small to capital or vice versa by choosing a **mask** and a proper **bit operation**. For example, the **a** letter has this binary representation `01(1)00001` while its capital counterpart has `01(0)00001`. They differ solely in the bit in parenthesis. In this case, converting the **a** letter from small to capital is basically setting the bit in parenthesis to one. To do so, we do the following:

```
/****************************************
convert small letter to captial letter.
========================================
     a: 01100001
  mask: 11011111  <-- (0xDF)  11(0)11111
      :---------
a&mask: 01000001  <-- A letter
*****************************************/
```

The code for converting a letter to A letter is

```
#include <cstdio>

int main()
{
    char op1 = 'a';  // "a" letter (i.e. small case)
    int mask = 0xDF; // choosing a proper mask

    printf("a (AND) mask = A\n");
    printf("%c   &   0xDF = %c\n", op1, op1 & mask);

    return 0;
}
```

The result is

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a   &   0xDF = A
```

Chapter 7: Bit fields

Bit fields tightly pack C and C++ structures to reduce size. This appears painless: specify the number of bits for members, and compiler does the work of co-mingling bits. The restriction is inability to take the address of a bit field member, since it is stored co-mingled. `sizeof()` is also disallowed.

The cost of bit fields is slower access, as memory must be retrieved and bitwise operations applied to extract or modify member values. These operations also add to executable size.

Section 7.1: Declaration and Usage

```
struct FileAttributes
{
    unsigned int ReadOnly: 1;
    unsigned int Hidden: 1;
};
```

Here, each of these two fields will occupy 1 bit in memory. It is specified by **`: 1`** expression after the variable names. Base type of bit field could be any integral type (8-bit int to 64-bit int). Using `unsigned` type is recommended, otherwise surprises may come.

If more bits are required, replace "1" with number of bits required. For example:

```
struct Date
{
    unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time
    unsigned int Month: 4;  // 2^4 = 16, enough to represent 1-12 month values.
    unsigned int Day:   5;  // 32
};
```

The whole structure is using just 22 bits, and with normal compiler settings, `sizeof` this structure would be 4 bytes.

Usage is pretty simple. Just declare the variable, and use it like ordinary structure.

```
Date d;

d.Year = 2016;
d.Month = 7;
d.Day =  22;

std::cout << "Year:" << d.Year << std::endl <<
        "Month:" << d.Month << std::endl <<
        "Day:" << d.Day << std::endl;
```

Chapter 8: Arrays

Arrays are elements of the same type placed in adjoining memory locations. The elements can be individually referenced by a unique identifier with an added index.

This allows you to declare multiple variable values of a specific type and access them individually without needing to declare a variable for each value.

Section 8.1: Array initialization

An array is just a block of sequential memory locations for a specific type of variable. Arrays are allocated the same way as normal variables, but with square brackets appended to its name `[]` that contain the number of elements that fit into the array memory.

The following example of an array uses the typ `int`, the variable name `arrayOfInts`, and the number of elements `[5]` that the array has space for:

```
int arrayOfInts[5];
```

An array can be declared and initialized at the same time like this

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

When initializing an array by listing all of its members, it is not necessary to include the number of elements inside the square brackets. It will be automatically calculated by the compiler. In the following example, it's 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

It is also possible to initialize only the first elements while allocating more space. In this case, defining the length in brackets is mandatory. The following will allocate an array of length 5 with partial initialization, the compiler initializes all remaining elements with the standard value of the element type, in this case zero.

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

Arrays of other basic data types may be initialized in the same way.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize

char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize

double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};

string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!"};
```

It is also important to take note that when accessing array elements, the array's element index(or position) starts from 0.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/};
std::cout << array[4]; //outputs 50
std::cout << array[0]; //outputs 10
```

Section 8.2: A fixed size raw array matrix (that is, a 2D raw array)

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   n_rows  = 3;
    int const   n_cols  = 7;
    int const   m[n_rows][n_cols] =             // A raw array matrix.
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];       // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Output:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

C++ doesn't support special syntax for indexing a multi-dimensional array. Instead such an array is viewed as an array of arrays (possibly of arrays, and so on), and the ordinary single index notation `[i]` is used for each level. In the example above `m[y]` refers to row `y` of `m`, where `y` is a zero-based index. Then this row can be indexed in turn, e.g. `m[y][x]`, which refers to the `x`th item – or column – of row `y`.

I.e. the last index varies fastest, and in the declaration the range of this index, which here is the number of columns per row, is the last and “innermost” size specified.

Since C++ doesn't provide built-in support for dynamic size arrays, other than dynamic allocation, a dynamic size matrix is often implemented as a class. Then the raw array matrix indexing notation `m[y][x]` has some cost, either by exposing the implementation (so that e.g. a view of a transposed matrix becomes practically impossible) or by adding some overhead and slight inconvenience when it's done by returning a proxy object from `operator[]`. And so the indexing notation for such an abstraction can and will usually be different, both in look-and-feel and in the order of indices, e.g. `m(x,y)` or `m.at(x,y)` or `m.item(x,y)`.

Section 8.3: Dynamically sized raw array

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>            // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }
auto main()
    -> int
{
    cout << "Sorting n integers provided by you.\\n";
    cout << "n? ";
    int const   n   = int_from( cin );
    int*        a   = new int[n];       // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\\n';

    delete[] a;
}
```

A program that declares an array `T a[n];` where `n` is determined a run-time, can compile with certain compilers that support C99 *variadic length arrays* (VLAs) as a language extension. But VLAs are not supported by standard C++. This example shows how to manually allocate a dynamic size array via a `new[]`-expression,

```
int*        a   = new int[n];       // ← Allocation of array of n items.
```

… then use it, and finally deallocate it via a `delete[]`-expression:

```
delete[] a;
```

The array allocated here has indeterminate values, but it can be zero-initialized by just adding an empty parenthesis `()`, like this: `new int[n]()`. More generally, for arbitrary item type, this performs a *value-initialization*.

As part of a function down in a call hierarchy this code would not be exception safe, since an exception before the `delete[]` expression (and after the `new[]`) would cause a memory leak. One way to address that issue is to automate the cleanup via e.g. a `std::unique_ptr` smart pointer. But a generally better way to address it is to just use a `std::vector`: that's what `std::vector` is there for.

Section 8.4: Array size: type safe at compile time

```
#include      // size_t, ptrdiff_t

//----------------------------------- Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
-> Size
{ return n; }


//----------------------------------- Usage:

#include
using namespace std;
auto main()
```

```
-> int
{
int const   a[]     = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
Size const  n       = n_items( a );
int         b[n]    = {};       // An array of the same size as a.

(void) b;
cout <}
```

The C idiom for array size, `sizeof(a)/sizeof(a[0])`, will accept a pointer as argument and will then generally yield an incorrect result.

For C++11

using C++11 you can do:

```
std::extent<decltype(MyArray)>::value;
```

Example:

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```

Up till C++17 (forthcoming as of this writing) C++ had no built-in core language or standard library utility to obtain the size of an array, but this can be implemented by passing the array *by reference* to a function template, as shown above. Fine but important point: the template size parameter is a `size_t`, somewhat inconsistent with the signed `Size` function result type, in order to accommodate the g++ compiler which sometimes insists on `size_t` for template matching.

With C++17 and later one may instead use `std::size`, which is specialized for arrays.

Section 8.5: Expanding dynamic size array by using std::vector

```
// Example of std::vector as an expanding dynamic size array.
#include <algorithm>            // std::sort
#include <iostream>
#include <vector>               // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a;      // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); }  // Expands as necessary.
    }

    sort( a.begin(), a.end() );
    int const n = a.size();
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';
}
```

`std::vector` is a standard library class template that provides the notion of a variable size array. It takes care of all the memory management, and the buffer is contiguous so a pointer to the buffer (e.g. `&v[0]` or `v.data()`) can be passed to API functions requiring a raw array. A `vector` can even be expanded at run time, via e.g. the `push_back` member function that appends an item.

The complexity of the sequence of *n* `push_back` operations, including the copying or moving involved in the vector expansions, is amortized O(*n*). "Amortized": on average.

Internally this is usually achieved by the vector *doubling* its buffer size, its capacity, when a larger buffer is needed. E.g. for a buffer starting out as size 1, and being repeatedly doubled as needed for *n*=17 `push_back` calls, this involves 1 + 2 + 4 + 8 + 16 = 31 copy operations, which is less than 2×*n* = 34. And more generally the sum of this sequence can't exceed 2×*n*.

Compared to the dynamic size raw array example, this `vector`-based code does not require the user to supply (and know) the number of items up front. Instead the vector is just expanded as necessary, for each new item value specified by the user.

Section 8.6: A dynamic size matrix using std::vector for storage

Unfortunately as of C++14 there's no dynamic size matrix class in the C++ standard library. Matrix classes that support dynamic size are however available from a number of 3rd party libraries, including the Boost Matrix library (a sub-library within the Boost library).

If you don't want a dependency on Boost or some other library, then one poor man's dynamic size matrix in C++ is just like

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

… where `vector` is `std::vector`. The matrix is here created by copying a row vector *n* times where *n* is the number of rows, here 3. It has the advantage of providing the same `m[y][x]` indexing notation as for a fixed size raw array matrix, but it's a bit inefficient because it involves a dynamic allocation for each row, and it's a bit unsafe because it's possible to inadvertently resize a row.

A more safe and efficient approach is to use a single vector as *storage* for the matrix, and map the client code's (*x*, *y*) to a corresponding index in that vector:

```
// A dynamic size matrix using std::vector for storage.

//--------------------------------------------- Machinery:
#include         // std::copy
#include          // assert
#include  // std::initializer_list
#include            // std::vector
#include          // ptrdiff_t

namespace my {
using Size = ptrdiff_t;
using std::initializer_list;
using std::vector;
```

```
template< class Item >
class Matrix
{
private:
vector    items_;
Size            n_cols_;

auto index_for( Size const x, Size const y ) const
-> Size
{ return y*n_cols_ + x; }

public:
auto n_rows() const -> Size { return items_.size()/n_cols_; }
auto n_cols() const -> Size { return n_cols_; }

auto item( Size const x, Size const y )
-> Item&
{ return items_[index_for(x, y)]; }

auto item( Size const x, Size const y ) const
-> Item const&
{ return items_[index_for(x, y)]; }

Matrix(): n_cols_( 0 ) {}

Matrix( Size const n_cols, Size const n_rows )
: items_( n_cols*n_rows )
, n_cols_( n_cols )
{}

Matrix( initializer_list< initializer_list > const& values )
: items_()
, n_cols_( values.size() == 0? 0 : values.begin()->size() )
{
for( auto const& row : values )
{
assert( Size( row.size() ) == n_cols_ );
items_.insert( items_.end(), row.begin(), row.end() );
}
}
};
}  // namespace my

//-------------------------------------------------- Usage:
using my::Matrix;

auto some_matrix()
-> Matrix
{
return
{
{  1,  2,  3,  4,  5,  6,  7 },
{  8,  9, 10, 11, 12, 13, 14 },
{ 15, 16, 17, 18, 19, 20, 21 }
};
}

#include
#include
using namespace std;
auto main() -> int
{
Matrix const m = some_matrix();
assert( m.n_cols() == 7 );
assert( m.n_rows() == 3 );
for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
{
for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
{
cout <← Note: not `m[y][x]`!
}
cout <}
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

The above code is not industrial grade: it's designed to show the basic principles, and serve the needs of students learning C++.

For example, one may define `operator()` overloads to simplify the indexing notation.

Chapter 9: Iterators

Section 9.1: Overview

Iterators are Positions

Iterators are a means of navigating and operating on a sequence of elements and are a generalized extension of pointers. Conceptually it is important to remember that iterators are positions, not elements. For example, take the following sequence:

```
A B C
```

The sequence contains three elements and four positions

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

Elements are things within a sequence. Positions are places where meaningful operations can happen to the sequence. For example, one inserts into a position, *before* or *after* element A, not into an element. Even deletion of an element (`erase(A)`) is done by first finding its position, then deleting it.

From Iterators to Values

To convert from a position to a value, an iterator is *dereferenced*:

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

One can think of an iterator as dereferencing to the value it refers to in the sequence. This is especially useful in understanding why you should never dereference the `end()` iterator in a sequence:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
↑           ↑
|           +-- An iterator here has no value. Do not dereference it!
+-------------- An iterator here dereferences to the value A.
```

In all the sequences and containers found in the C++ standard library, `begin()` will return an iterator to the first position, and `end()` will return an iterator to one past the last position (*not* the last position!). Consequently, the names of these iterators in algorithms are oftentimes labelled `first` and `last`:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
↑           ↑
|           |
+- first    +- last
```

It is also possible to obtain an iterator to *any sequence*, because even an empty sequence contains at least one position:

```
+---+
|   |
+---+
```

In an empty sequence, `begin()` and `end()` will be the same position, and *neither* can be dereferenced:

```
+---+
|   |
+---+
  ↑
  |
  +- empty_sequence.begin()
  |
  +- empty_sequence.end()
```

The alternative visualization of iterators is that they mark the positions *between* elements:

```
+---+---+---+
| A | B | C |
+---+---+---+
↑   ^   ^   ↑
|           |
+- first    +- last
```

and dereferencing an iterator returns a reference to the element coming after the iterator. Some situations where this view is particularly useful are:

- `insert` operations will insert elements into the position indicated by the iterator,
- `erase` operations will return an iterator corresponding to the same position as the one passed in,
- an iterator and its corresponding reverse iterator are located in the same .position between elements

Invalid Iterators

An iterator becomes *invalidated* if (say, in the course of an operation) its position is no longer a part of a sequence. An invalidated iterator cannot be dereferenced until it has been reassigned to a valid position. For example:

```
std::vector<int>::iterator first;
{
    std::vector<int> foo;
    first = foo.begin(); // first is now valid
} // foo falls out of scope and is destroyed
// At this point first is now invalid
```

The many algorithms and sequence member functions in the C++ standard library have rules governing when iterators are invalidated. Each algorithm is different in the way they treat (and invalidate) iterators.

Navigating with Iterators

As we know, iterators are for navigating sequences. In order to do that an iterator must migrate its position throughout the sequence. Iterators can advance forward in the sequence and some can advance backwards:

```
auto first = my_vector.begin();
++first;                                              // advance the iterator 1 position
```

```
std::advance(first, 1);                              // advance the iterator 1 position
first = std::next(first);                            // returns iterator to the next element
std::advance(first, -1);                             // advance the iterator 1 position backwards
first = std::next(first, 20);                        // returns iterator to the element 20 position
forward
first = std::prev(first, 5);                         // returns iterator to the element 5 position
backward
auto dist = std::distance(my_vector.begin(), first); // returns distance between two iterators.
```

Note, second argument of std::distance should be reachable from the first one(or, in other words `first` should be less or equal than `second`).

Even though you can perform arithmetic operators with iterators, not all operations are defined for all types of iterators. `a = b + 3;` would work for Random Access Iterators, but wouldn't work for Forward or Bidirectional Iterators, which still can be advanced by 3 position with something like `b = a; ++b; ++b; ++b;`. So it is recommended to use special functions in case you are not sure what is iterator type (for example, in a template function accepting iterator).

Iterator Concepts

The C++ standard describes several different iterator concepts. These are grouped according to how they behave in the sequences they refer to. If you know the concept an iterator *models* (behaves like), you can be assured of the behavior of that iterator *regardless of the sequence to which it belongs*. They are often described in order from the most to least restrictive (because the next iterator concept is a step better than its predecessor):

- Input Iterators : Can be dereferenced *only once* per position. Can only advance, and only one position at a time.
- Forward Iterators : An input iterator that can be dereferenced any number of times.
- Bidirectional Iterators : A forward iterator that can also advance *backwards* one position at a time.
- Random Access Iterators : A bidirectional iterator that can advance forwards or backwards any number of positions at a time.
- Contiguous Iterators (since C++17) : A random access iterator that guaranties that underlying data is contiguous in memory.

Algorithms can vary depending on the concept modeled by the iterators they are given. For example, although `random_shuffle` can be implemented for forward iterators, a more efficient variant that requires random access iterators could be provided.

Iterator traits

Iterator traits provide uniform interface to the properties of iterators. They allow you to retrieve value, difference, pointer, reference types and also category of iterator:

```
template<class Iter>
Iter find(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type val)  {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}
```

Category of iterator can be used to specialize algorithms:

```
template<class BidirIt>
```

```
void test(BidirIt a, std::bidirectional_iterator_tag)  {
    std::cout << "Bidirectional iterator is used" << std::endl;
}

template<class ForwIt>
void test(ForwIt a, std::forward_iterator_tag)  {
    std::cout << "Forward iterator is used" << std::endl;
}

template<class Iter>
void test(Iter a)  {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}
```

Categories of iterators are basically iterators concepts, except Contiguous Iterators don't have their own tag, since it was found to break code.

Section 9.2: Vector Iterator

`begin` returns an `iterator` to the first element in the sequence container.

`end` returns an `iterator` to the first element past the end.

If the vector object is `const`, both `begin` and `end` return a `const_iterator`. If you want a `const_iterator` to be returned even if your vector is not `const`, you can use `cbegin` and `cend`.

Example:

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

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };  //intialize vector using an initializer_list

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Output:

> 1 2 3 4 5

Section 9.3: Map Iterator

An iterator to the first element in the container.

If a map object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

```
// Create a map and insert some values
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
mymap['c'] = 300;
```

```
// Iterate over all tuples
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

Output:

> a => 200
> b => 100
> c => 300

Section 9.4: Reverse Iterators

If we want to iterate backwards through a list or vector we can use a `reverse_iterator`. A reverse iterator is made from a bidirectional, or random access iterator which it keeps as a member which can be accessed through `base()`.

To iterate backwards use `rbegin()` and `rend()` as the iterators for the end of the collection, and the start of the collection respectively.

For instance, to iterate backwards use:

```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // prints 54321
```

A reverse iterator can be converted to a forward iterator via the `base()` member function. The relationship is that the reverse iterator references one element past the `base()` iterator:

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&*r == &*(i-1)); // always true if r, (i-1) are dereferenceable
                        // and are not proxy iterators

 +---+---+---+---+---+---+---+
 |   | 1 | 2 | 3 | 4 | 5 |   |
 +---+---+---+---+---+---+---+
   ↑   ↑               ↑   ↑
   |   |               |   |
rend() |         rbegin()  end()
       |                   rbegin().base()
     begin()
     rend().base()
```

In the visualization where iterators mark positions between elements, the relationship is simpler:

```

  +---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
↑                   ↑
|                   |
|                 end()
|                 rbegin()
begin()             rbegin().base()
rend()
rend().base()
```

Section 9.5: Stream Iterators

Stream iterators are useful when we need to read a sequence or print formatted data from a container:

```
// Data stream. Any number of various whitespace characters will be OK.
std::istringstream istr("1\t 2     3 4");
std::vector<int> v;

// Constructing stream iterators and copying data from stream into vector.
std::copy(
    // Iterator which will read stream data as integers.
    std::istream_iterator<int>(istr),
    // Default constructor produces end-of-stream iterator.
    std::istream_iterator<int>(),
    std::back_inserter(v));

// Print vector contents.
std::copy(v.begin(), v.end(),
    //Will print values to standard output as integers delimeted by " -- ".
    std::ostream_iterator<int>(std::cout, " -- "));
```

The example program will print `1 -- 2 -- 3 -- 4 --` to standard output.

Section 9.6: C Iterators (Pointers)

```
// This creates an array with 5 values.
const int array[] = { 1, 2, 3, 4, 5 };

#ifdef BEFORE_CPP11

// You can use `sizeof` to determine how many elements are in an array.
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);

// Then you can iterate over the array by incrementing a pointer until
// it reaches past the end of our array.
for (const int* i = first; i < afterLast; ++i) {
    std::cout << *i << std::endl;
}

#else

// With C++11, you can let the STL compute the start and end iterators:
for (auto i = std::begin(array); i != std::end(array); ++i) {
    std::cout << *i << std::endl;
}

#endif
```

This code would output the numbers 1 through 5, one on each line like this:

> 1
> 2
> 3
> 4

Breaking It Down

```
const int array[] = { 1, 2, 3, 4, 5 };
```

This line creates a new integer array with 5 values. C arrays are just pointers to memory where each value is stored together in a contiguous block.

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

These lines create two pointers. The first pointer is given the value of the array pointer, which is the address of the first element in the array. The `sizeof` operator when used on a C array returns the size of the array in bytes. Divided by the size of an element this gives the number of elements in the array. We can use this to find the address of the block *after* the array.

```
for (const int* i = first; i < afterLast; ++i) {
```

Here we create a pointer which we will use as an iterator. It is initialized with the address of the first element we want to iterate over, and we'll continue to iterate as long as `i` is less than `afterLast`, which means as long as `i` is pointing to an address within `array`.

```
    std::cout << *i << std::endl;
```

Finally, within the loop we can access the value our iterator `i` is pointing to by dereferencing it. Here the dereference operator `*` returns the value at the address in `i`.

Section 9.7: Write your own generator-backed iterator

A common pattern in other languages is having a function that produces a "stream" of objects, and being able to use loop-code to loop over it.

We can model this in C++ as

```
template<class T>
struct generator_iterator {
  using difference_type=std::ptrdiff_t;
  using value_type=T;
  using pointer=T*;
  using reference=T;
  using iterator_category=std::input_iterator_tag;
  std::optional<T> state;
  std::function< std::optional<T>() > operation;
  // we store the current element in "state" if we have one:
  T operator*() const {
    return *state;
  }
  // to advance, we invoke our operation.  If it returns a nullopt
  // we have reached the end:
  generator_iterator& operator++() {
    state = operation();
    return *this;
  }
  generator_iterator operator++(int) {
    auto r = *this;
    ++(*this);
    return r;
  }
  // generator iterators are only equal if they are both in the "end" state:
  friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
    if (!lhs.state && !rhs.state) return true;
    return false;
  }
  friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
    return !(lhs==rhs);
  }
  // We implicitly construct from a std::function with the right signature:
  generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
  {
    if (operation)
      state = operation();
  }
  // default all special member functions:
  generator_iterator( generator_iterator && ) =default;
  generator_iterator( generator_iterator const& ) =default;
  generator_iterator& operator=( generator_iterator && ) =default;
  generator_iterator& operator=( generator_iterator const& ) =default;
  generator_iterator() =default;
};
```

live example.

We store the generated element early so we can more easily detect if we are already at the end.

As the function of an end generator iterator is never used, we can create a range of generator iterators by only copying the `std::function` once. A default constructed generator iterator compares equal to itself, and to all other end-generator-iterators.

Chapter 10: Basic input/output in c++

Section 10.1: user input and standard output

```
#include <iostream>

int main()
{
    int value;
    std::cout << "Enter a value: " << std::endl;
    std::cin >> value;
    std::cout << "The square of entered value is: " << value * value << std::endl;
    return 0;
}
```

Chapter 11: Loops

A loop statement executes a group of statements repeatedly until a condition is met. There are 3 types of primitive loops in C++: for, while, and do...while.

Section 11.1: Range-Based For

Version ≥ C++11

`for` loops can be used to iterate over the elements of a iterator-based range, without using a numeric index or directly accessing the iterators:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(auto val: v)
{
    std::cout << val << " ";
}

std::cout << std::endl;
```

This will iterate over every element in `v`, with `val` getting the value of the current element. The following statement:

```
for (for-range-declaration : for-range-initializer ) statement
```

is equivalent to:

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr, __end = end-expr;
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Version ≥ C++17

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr;
    auto __end = end-expr; // end is allowed to be a different type than begin in C++17
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

This change was introduced for the planned support of Ranges TS in C++20.

In this case, our loop is equivalent to:

```
{
    auto&& __range = v;
    auto __begin = v.begin(), __end = v.end();
    for (; __begin != __end; ++__begin) {
        auto val = *__begin;
        std::cout << val << " ";
    }
}
```

Note that `auto val` declares a value type, which will be a copy of a value stored in the range (we are copy-initializing it from the iterator as we go). If the values stored in the range are expensive to copy, you may want to use `const auto &val`. You are also not required to use `auto`; you can use an appropriate typename, so long as it is implicitly convertible from the range's value type.

If you need access to the iterator, range-based for cannot help you (not without some effort, at least).

If you wish to reference it, you may do so:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(float &val: v)
{
    std::cout << val << " ";
}
```

You could iterate on `const` reference if you have `const` container:

```
const vector<float> v = {0.4f, 12.5f, 16.234f};

for(const float &val: v)
{
    std::cout << val << " ";
}
```

One would use forwarding references when the sequence iterator returns a proxy object and you need to operate on that object in a non-`const` way. Note: it will most likely confuse readers of your code.

```
vector<bool> v(10);

for(auto&& val: v)
{
    val = true;
}
```

The "range" type provided to range-based `for` can be one of the following:

- Language arrays:

  ```
  float arr[] = {0.4f, 12.5f, 16.234f};

  for(auto val: arr)
  {
      std::cout << val << " ";
  }
  ```

  Note that allocating a dynamic array does not count:

  ```
  float *arr = new float[3]{0.4f, 12.5f, 16.234f};

  for(auto val: arr) //Compile error.
  {
      std::cout << val << " ";
  }
  ```

- Any type which has member functions `begin()` and `end()`, which return iterators to the elements of the type. The standard library containers qualify, but user-defined types can be used as well:

```
struct Rng
{
    float arr[3];

    // pointers are iterators
    const float* begin() const {return &arr[0];}
    const float* end() const   {return &arr[3];}
    float* begin() {return &arr[0];}
    float* end()   {return &arr[3];}
};

int main()
{
    Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

- Any type which has non-member `begin(type)` and `end(type)` functions which can found via argument dependent lookup, based on `type`. This is useful for creating a range type without having to modify class type itself:

```
namespace Mine
{
    struct Rng {float arr[3];};

    // pointers are iterators
    const float* begin(const Rng &rng) {return &rng.arr[0];}
    const float* end(const Rng &rng) {return &rng.arr[3];}
    float* begin(Rng &rng) {return &rng.arr[0];}
    float* end(Rng &rng) {return &rng.arr[3];}
}

int main()
{
    Mine::Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

Section 11.2: For loop

A `for` loop executes statements in the `loop body`, while the loop `condition` is true. Before the loop `initialization statement` is executed exactly once. After each cycle, the `iteration execution` part is executed.

A `for` loop is defined as follows:

```
for (/*initialization statement*/; /*condition*/; /*iteration execution*/)
```

```
{
    // body of the loop
}
```

Explanation of the placeholder statements:

- `initialization statement`: This statement gets executed only once, at the beginning of the `for` loop. You can enter a declaration of multiple variables of one type, such as `int i = 0, a = 2, b = 3`. These variables are only valid in the scope of the loop. Variables defined before the loop with the same name are hidden during execution of the loop.
- `condition`: This statement gets evaluated ahead of each *loop body* execution, and aborts the loop if it evaluates to `false`.
- `iteration execution`: This statement gets executed after the loop *body*, ahead of the next *condition* evaluation, unless the `for` loop is aborted in the *body* (by `break`, `goto`, `return` or an exception being thrown). You can enter multiple statements in the `iteration execution` part, such as `a++, b+=10, c=b+a`.

The rough equivalent of a `for` loop, rewritten as a `while` loop is:

```
/*initialization*/
while (/*condition*/)
{
    // body of the loop; using 'continue' will skip to increment part below
    /*iteration execution*/
}
```

The most common case for using a `for` loop is to execute statements a specific number of times. For example, consider the following:

```
for(int i = 0; i < 10; i++) {
    std::cout << i << std::endl;
}
```

A valid loop is also:

```
for(int a = 0, b = 10, c = 20; (a+b+c < 100); c--, b++, a+=c) {
    std::cout << a << " " << b << " " << c << std::endl;
}
```

An example of hiding declared variables before a loop is:

```
int i = 99; //i = 99
for(int i = 0; i < 10; i++) { //we declare a new variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 99
```

But if you want to use the already declared variable and not hide it, then omit the declaration part:

```
int i = 99; //i = 99
for(i = 0; i < 10; i++) { //we are using already declared variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 10
```

Notes:

- The initialization and increment statements can perform operations unrelated to the condition statement, or nothing at all - if you wish to do so. But for readability reasons, it is best practice to only perform operations directly relevant to the loop.
- A variable declared in the initialization statement is visible only inside the scope of the `for` loop and is released upon termination of the loop.
- Don't forget that the variable which was declared in the `initialization statement` can be modified during the loop, as well as the variable checked in the `condition`.

Example of a loop which counts from 0 to 10:

```
for (int counter = 0; counter <= 10; ++counter)
{
    std::cout << counter << '\n';
}
// counter is not accessible here (had value 11 at the end)
```

Explanation of the code fragments:

- `int counter = 0` initializes the variable `counter` to 0. (This variable can only be used inside of the `for` loop.)
- `counter <= 10` is a Boolean condition that checks whether `counter` is less than or equal to 10. If it is `true`, the loop executes. If it is `false`, the loop ends.
- `++counter` is an increment operation that increments the value of `counter` by 1 ahead of the next condition check.

By leaving all statements empty, you can create an infinite loop:

```
// infinite loop
for (;;)
    std::cout << "Never ending!\n";
```

The `while` loop equivalent of the above is:

```
// infinite loop
while (true)
    std::cout << "Never ending!\n";
```

However, an infinite loop can still be left by using the statements `break`, `goto`, or `return` or by throwing an exception.

The next common example of iterating over all elements from an STL collection (e.g., a `vector`) without using the **`<algorithm>`** header is:

```
std::vector<std::string> names = {"Albert Einstein", "Stephen Hawking", "Michael Ellis"};
for(std::vector<std::string>::iterator it = names.begin(); it != names.end(); ++it) {
    std::cout << *it << std::endl;
}
```

Section 11.3: While loop

A `while` loop executes statements repeatedly until the given condition evaluates to `false`. This control statement is used when it is not known, in advance, how many times a block of code is to be executed.

For example, to print all the numbers from 0 up to 9, the following code can be used:

```
int i = 0;
```

```
while (i < 10)
{
    std::cout << i << " ";
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; "0 1 2 3 4 5 6 7 8 9" is printed to the console
```

Version ≥ C++17

Note that since C++17, the first 2 statements can be combined

```
while (int i = 0; i < 10)
//... The rest is the same
```

To create an infinite loop, the following construct can be used:

```
while (true)
{
    // Do something forever (however, you can exit the loop by calling 'break'
}
```

There is another variant of `while` loops, namely the `do...while` construct. See the do-while loop example for more information.

Section 11.4: Do-while loop

A *do-while* loop is very similar to a *while* loop, except that the condition is checked at the end of each cycle, not at the start. The loop is therefore guaranteed to execute at least once.

The following code will print `0`, as the condition will evaluate to `false` at the end of the first iteration:

```
int i =0;
do
{
    std::cout << i;
    ++i; // Increment counter
}
while (i < 0);
std::cout << std::endl; // End of line; 0 is printed to the console
```

Note: Do not forget the semicolon at the end of `while(condition);`, which is needed in the *do-while* construct.

In contrast to the *do-while* loop, the following will not print anything, because the condition evaluates to `false` at the beginning of the first iteration:

```
int i =0;
while (i < 0)
{
    std::cout << i;
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; nothing is printed to the console
```

Note: A *while* loop can be exited without the condition becoming false by using a `break`, `goto`, or `return` statement.

```
int i = 0;
do
{
    std::cout << i;
    ++i; // Increment counter
    if (i > 5)
    {
        break;
    }
}
while (true);
std::cout << std::endl; // End of line; 0 1 2 3 4 5 is printed to the console
```

A trivial *do-while* loop is also occasionally used to write macros that require their own scope (in which case the trailing semicolon is omitted from the macro definition and required to be provided by the user):

```
#define BAD_MACRO(x) f1(x); f2(x); f3(x);

// Only the call to f1 is protected by the condition here
if (cond) BAD_MACRO(var);

#define GOOD_MACRO(x) do { f1(x); f2(x); f3(x); } while(0)

// All calls are protected here
if (cond) GOOD_MACRO(var);
```

Section 11.5: Loop Control statements : Break and Continue

Loop control statements are used to change the flow of execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. The `break` and `continue` are loop control statements.

The `break` statement terminates a loop without any further consideration.

```
for (int i = 0; i < 10; i++)
{
    if (i == 4)
        break; // this will immediately exit our loop
    std::cout << i << '\n';
}
```

The above code will print out:

```
1
2
3
```

The `continue` statement does not immediately exit the loop, but rather skips the rest of the loop body and goes to the top of the loop (including checking the condition).

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // evaluates to true if i is even
        continue; // this will immediately go back to the start of the loop
    /* the next line will only be reached if the above "continue" statement
       does not execute  */
    std::cout << i << " is an odd number\n";
}
```

The above code will print out:

```
1 is an odd number
3 is an odd number
5 is an odd number
```

Because such control flow changes are sometimes difficult for humans to easily understand, `break` and `continue` are used sparingly. More straightforward implementation are usually easier to read and understand. For example, the first `for` loop with the `break` above might be rewritten as:

```
for (int i = 0; i < 4; i++)
{
    std::cout << i << '\n';
}
```

The second example with `continue` might be rewritten as:

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 != 0) {
        std::cout << i << " is an odd number\n";
    }
}
```

Section 11.6: Declaration of variables in conditions

In the condition of the `for` and `while` loops, it's also permitted to declare an object. This object will be considered to be in scope until the end of the loop, and will persist through each iteration of the loop:

```
for (int i = 0; i < 5; ++i) {
    do_something(i);
}
// i is no longer in scope.

for (auto& a : some_container) {
    a.do_something();
}
// a is no longer in scope.

while(std::shared_ptr<Object> p = get_object()) {
   p->do_something();
}
// p is no longer in scope.
```

However, it is not permitted to do the same with a `do...while` loop; instead, declare the variable before the loop, and (optionally) enclose both the variable and the loop within a local scope if you want the variable to go out of scope after the loop ends:

```
//This doesn't compile
do {
    s = do_something();
} while (short s > 0);

// Good
short s;
do {
    s = do_something();
} while (s > 0);
```

This is because the *statement* portion of a `do...while` loop (the loop's body) is evaluated before the *expression* portion (the `while`) is reached, and thus, any declaration in the *expression* will not be visible during the first iteration of the loop.

Section 11.7: Range-for over a sub-range

Using range-base loops, you can loop over a sub-part of a given container or other range by generating a proxy object that qualifies for range-based for loops.

```
template<class Iterator, class Sentinel=Iterator>
struct range_t {
  Iterator b;
  Sentinel e;
  Iterator begin() const { return b; }
  Sentinel end() const { return e; }
  bool empty() const { return begin()==end(); }
  range_t without_front( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {std::next(b, count), e};
  }
  range_t without_back( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {b, std::prev(e, count)};
  }
};

template<class Iterator, class Sentinel>
range_t<Iterator, Sentinel> range( Iterator b, Sentinal e ) {
  return {b,e};
}
template<class Iterable>
auto range( Iterable& r ) {
  using std::begin; using std::end;
  return range(begin(r),end(r));
}

template<class C>
auto except_first( C& c ) {
  auto r = range(c);
  if (r.empty()) return r;
  return r.without_front();
}
```

now we can do:

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

for (auto i : except_first(v))
  std::cout << i << '\n';
```

and print out

```
2
```

```
3
4
```

Be aware that intermediate objects generated in the `for(:range_expression)` part of the `for` loop will have expired by the time the `for` loop starts.

Chapter 12: File I/O

C++ file I/O is done via *streams*. The key abstractions are:

`std::istream` for reading text.

`std::ostream` for writing text.

`std::streambuf` for reading or writing characters.

Formatted input uses `operator>>`.

Formatted output uses `operator<<`.

Streams use `std::locale`, e.g., for details of the formatting and for translation between external encodings and the internal encoding.

More on streams: <iostream> Library

Section 12.1: Writing to a file

There are several ways to write to a file. The easiest way is to use an output file stream (`ofstream`) together with the stream insertion operator (`<<`):

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

Instead of `<<`, you can also use the output file stream's member function `write()`:

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";

    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

After writing to a stream, you should always check if error state flag `badbit` has been set, as it indicates whether the operation failed or not. This can be done by calling the output file stream's member function `bad()`:

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

Section 12.2: Opening a file

Opening a file is done in the same way for all 3 file streams (`ifstream`, `ofstream`, and `fstream`).

You can open the file directly in the constructor:

```
std::ifstream ifs("foo.txt");  // ifstream: Opens file "foo.txt" for reading only.

std::ofstream ofs("foo.txt");  // ofstream: Opens file "foo.txt" for writing only.
```

```
std::fstream iofs("foo.txt");  // fstream:  Opens file "foo.txt" for reading and writing.
```

Alternatively, you can use the file stream's member function `open()`:

```
std::ifstream ifs;
ifs.open("bar.txt");           // ifstream: Opens file "bar.txt" for reading only.

std::ofstream ofs;
ofs.open("bar.txt");           // ofstream: Opens file "bar.txt" for writing only.

std::fstream iofs;
iofs.open("bar.txt");          // fstream:  Opens file "bar.txt" for reading and writing.
```

You should **always** check if a file has been opened successfully (even when writing). Failures can include: the file doesn't exist, file hasn't the right access rights, file is already in use, disk errors occurred, drive disconnected ... Checking can be done as follows:

```
// Try to read the file 'foo.txt'.
std::ifstream ifs("fooo.txt");  // Note the typo; the file can't be opened.

// Check if the file has been opened successfully.
if (!ifs.is_open()) {
    // The file hasn't been opened; take appropriate actions here.
    throw CustomException(ifs, "File could not be opened");
}
```

When file path contains backslashes (for example, on Windows system) you should properly escape them:

```
// Open the file 'c:\\folder\\foo.txt' on Windows.
std::ifstream ifs("c:\\\\folder\\\\foo.txt"); // using escaped backslashes
```

Version ≥ C++11

or use raw literal:

```
// Open the file 'c:\\folder\\foo.txt' on Windows.
std::ifstream ifs(R"(c:\\folder\\foo.txt)"); // using raw literal
```

or use forward slashes instead:

```
// Open the file 'c:\\folder\\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
```

Version ≥ C++11

If you want to open file with non-ASCII characters in path on Windows currently you can use **non-standard** wide character path argument:

```
// Open the file 'пример\\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\\foo.txt)"); // using wide characters with raw literal
```

Section 12.3: Reading from a file

There are several ways to read data from a file.

If you know how the data is formatted, you can use the stream extraction operator (`>>`). Let's assume you have a file named *foo.txt* which contains the following data:

```
John Doe 25 4 6 1987
Jane Doe 15 5 24 1976
```

Then you can use the following code to read that data from the file:

```
// Define variables.
std::ifstream is("foo.txt");
std::string firstname, lastname;
int age, bmonth, bday, byear;

// Extract firstname, lastname, age, bday month, bday day, and bday year in that order.
// Note: '>>' returns false if it reached EOF (end of file) or if the input data doesn't
// correspond to the type of the input variable (for example, the string "foo" can't be
// extracted into an 'int' variable).
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)
    // Process the data that has been read.
```

The stream extraction operator `>>` extracts every character and stops if it finds a character that can't be stored or if it is a special character:

- For string types, the operator stops at a whitespace () or at a newline (`\n`).
- For numbers, the operator stops at a non-number character.

This means that the following version of the file *foo.txt* will also be successfully read by the previous code:

```
John
Doe 25
4 6 1987


Jane
Doe
15 5
24
1976
```

The stream extraction operator `>>` always returns the stream given to it. Therefore, multiple operators can be chained together in order to read data consecutively. However, a stream can also be used as a Boolean expression (as shown in the `while` loop in the previous code). This is because the stream classes have a conversion operator for the type `bool`. This `bool()` operator will return `true` as long as the stream has no errors. If a stream goes into an error state (for example, because no more data can be extracted), then the `bool()` operator will return `false`. Therefore, the `while` loop in the previous code will be exited after the input file has been read to its end.

If you wish to read an entire file as a string, you may use the following code:

```
// Opens 'foo.txt'.
std::ifstream is("foo.txt");
std::string whole_file;

// Sets position to the end of the file.
is.seekg(0, std::ios::end);

// Reserves memory for the file.
whole_file.reserve(is.tellg());

// Sets position to the start of the file.
is.seekg(0, std::ios::beg);
```

```
// Sets contents of 'whole_file' to all characters in the file.
whole_file.assign(std::istreambuf_iterator<char>(is),
  std::istreambuf_iterator<char>());
```

This code reserves space for the `string` in order to cut down on unneeded memory allocations.

If you want to read a file line by line, you can use the function `getline()`:

```
std::ifstream is("foo.txt");

// The function getline returns false if there are no more lines.
for (std::string str; std::getline(is, str);) {
    // Process the line that has been read.
}
```

If you want to read a fixed number of characters, you can use the stream's member function `read()`:

```
std::ifstream is("foo.txt");
char str[4];

// Read 4 characters from the file.
is.read(str, 4);
```

After executing a read command, you should always check if the error state flag `failbit` has been set, as it indicates whether the operation failed or not. This can be done by calling the file stream's member function `fail()`:

```
is.read(str, 4); // This operation might fail for any reason.

if (is.fail())
    // Failed to read!
```

Section 12.4: Opening modes

When creating a file stream, you can specify an opening mode. An opening mode is basically a setting to control how the stream opens the file.

(All modes can be found in the `std::ios` namespace.)

An opening mode can be provided as second parameter to the constructor of a file stream or to its `open()` member function:

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);

std::ifstream is;
is.open("foo.txt", std::ios::in | std::ios::binary);
```

It is to be noted that you have to set `ios::in` or `ios::out` if you want to set other flags as they are not implicitly set by the iostream members although they have a correct default value.

If you don't specify an opening mode, then the following default modes are used:

- `ifstream` - `in`
- `ofstream` - `out`
- `fstream` - `in` and `out`

The file opening modes that you may specify by design are:

Mode	Meaning	For	Description
`app`	append	Output	Appends data at the end of the file.
`binary`	binary	Input/Output	Input and output is done in binary.
`in`	input	Input	Opens the file for reading.
`out`	output	Output	Opens the file for writing.
`trunc`	truncate	Input/Output	Removes contents of the file when opening.
`ate`	at end	Input	Goes to the end of the file when opening.

Note: Setting the `binary` mode lets the data be read/written exactly as-is; not setting it enables the translation of the newline `'\n'` character to/from a platform specific end of line sequence.

For example on Windows the end of line sequence is CRLF (`"\r\n"`).
Write: `"\n"` => `"\r\n"`
Read: `"\r\n"` => `"\n"`

Section 12.5: Reading an ASCII file into a std::string

```
std::ifstream f("file.txt");

if (f)
{
  std::stringstream buffer;
  buffer << f.rdbuf();
  f.close();

  // The content of "file.txt" is available in the string `buffer.str()`
}
```

The `rdbuf()` method returns a pointer to a `streambuf` that can be pushed into `buffer` via the `stringstream::operator<<` member function.

Another possibility (popularized in Effective STL by Scott Meyers) is:

```
std::ifstream f("file.txt");

if (f)
{
  std::string str((std::istreambuf_iterator<char>(f)),
                  std::istreambuf_iterator<char>());

  // Operations on `str`...
}
```

This is nice because requires little code (and allows reading a file directly into any STL container, not only strings) but can be slow for big files.

NOTE: the extra parentheses around the first argument to the string constructor are essential to prevent the *most vexing parse* problem.

Last but not least:

```
std::ifstream f("file.txt");

if (f)
{
  f.seekg(0, std::ios::end);
  const auto size = f.tellg();

  std::string str(size, ' ');
  f.seekg(0);
  f.read(&str[0], size);
  f.close();

  // Operations on `str`...
}
```

which is probably the fastest option (among the three proposed).

Section 12.6: Writing files with non-standard locale settings

If you need to write a file using different locale settings to the default, you can use `std::locale` and `std::basic_ios::imbue()` to do that for a specific file stream:

Guidance for use:

- You should always apply a local to a stream before opening the file.
- Once the stream has been imbued you should not change the locale.

Reasons for Restrictions: Imbuing a file stream with a locale has undefined behavior if the current locale is not state independent or not pointing at the beginning of the file.

UTF-8 streams (and others) are not state independent. Also a file stream with a UTF-8 locale may try and read the BOM marker from the file when it is opened; so just opening the file may read characters from the file and it will not be at the beginning.

```
#include <iostream>
#include <fstream>
#include <locale>

int main()
{
  std::cout << "User-preferred locale setting is "
            << std::locale("").name().c_str() << std::endl;

  // Write a floating-point value using the user's preferred locale.
  std::ofstream ofs1;
  ofs1.imbue(std::locale(""));
  ofs1.open("file1.txt");
  ofs1 << 78123.456 << std::endl;

  // Use a specific locale (names are system-dependent)
  std::ofstream ofs2;
  ofs2.imbue(std::locale("en_US.UTF-8"));
  ofs2.open("file2.txt");
  ofs2 << 78123.456 << std::endl;

  // Switch to the classic "C" locale
  std::ofstream ofs3;
  ofs3.imbue(std::locale::classic());
  ofs3.open("file3.txt");
  ofs3 << 78123.456 << std::endl;
}
```

Explicitly switching to the classic "C" locale is useful if your program uses a different default locale and you want to ensure a fixed standard for reading and writing files. With a "C" preferred locale, the example writes

```
78,123.456
78,123.456
78123.456
```

If, for example, the preferred locale is German and hence uses a different number format, the example writes

```
78 123,456
78,123.456
78123.456
```

(note the decimal comma in the first line).

Section 12.7: Checking end of file inside a loop condition, bad practice?

`eof` returns `true` only **after** reading the end of file. It does NOT indicate that the next read will be the end of stream.

```
while (!f.eof())
{
  // Everything is OK

  f >> buffer;

  // What if *only* now the eof / fail bit is set?

  /* Use `buffer` */
}
```

You could correctly write:

```
while (!f.eof())
{
  f >> buffer >> std::ws;

  if (f.fail())
    break;

  /* Use `buffer` */
}
```

but

```
while (f >> buffer)
{
  /* Use `buffer` */
}
```

is simpler and less error prone.

Further references:

- `std::ws`: discards leading whitespace from an input stream
- `std::basic_ios::fail`: returns `true` if an error has occurred on the associated stream

Section 12.8: Flushing a stream

File streams are buffered by default, as are many other types of streams. This means that writes to the stream may not cause the underlying file to change immediately. In oder to force all buffered writes to take place immediately, you can *flush* the stream. You can do this either directly by invoking the `flush()` method or through the `std::flush` stream manipulator:

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

There is a stream manipulator `std::endl` that combines writing a newline with flushing the stream:

```
// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;
```

Buffering can improve the performance of writing to a stream. Therefore, applications that do a lot of writing should avoid flushing unnecessarily. Contrary, if I/O is done infrequently, applications should consider flushing frequently in order to avoid data getting stuck in the stream object.

Section 12.9: Reading a file into a container

In the example below we use `std::string` and `operator>>` to read items from the file.

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v;

    std::string s;
    while(file >> s) // keep reading until we run out
    {
        v.push_back(s);
    }
```

In the above example we are simply iterating through the file reading one "item" at a time using `operator>>`. This same affect can be achieved using the `std::istream_iterator` which is an input iterator that reads one "item" at a time from the stream. Also most containers can be constructed using two iterators so we can simplify the above code to:

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v(std::istream_iterator<std::string>{file},
                                std::istream_iterator<std::string>{});
```

We can extend this to read any object types we like by simply specifying the object we want to read as the template parameter to the `std::istream_iterator`. Thus we can simply extend the above to read lines (rather than words) like this:

```
// Unfortunately there is  no built in type that reads line using >>
// So here we build a simple helper class to do it. That will convert
// back to a string when used in string context.
struct Line
```

```
{
    // Store data here
    std::string data;
    // Convert object to string
    operator std::string const&() const {return data;}
    // Read a line from a stream.
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};


    std::ifstream file("file3.txt");

    // Read the lines of a file into a container.
    std::vector<std::string>  v(std::istream_iterator<Line>{file},
                                std::istream_iterator<Line>{});
```

Section 12.10: Copying a file

```
std::ifstream  src("source_filename", std::ios::binary);
std::ofstream  dst("dest_filename",   std::ios::binary);
dst << src.rdbuf();
```

Version ≥ C++17

With C++17 the standard way to copy a file is including the **`<filesystem>`** header and using `copy_file`:

```
std::fileystem::copy_file("source_filename", "dest_filename");
```

The filesystem library was originally developed as `boost.filesystem` and finally merged to ISO C++ as of C++17.

Section 12.11: Closing a file

Explicitly closing a file is rarely necessary in C++, as a file stream will automatically close its associated file in its destructor. However, you should try to limit the lifetime of a file stream object, so that it does not keep the file handle open longer than necessary. For example, this can be done by putting all file operations into an own scope (`{}`):

```
std::string const prepared_data = prepare_data();
{
    // Open a file for writing.
    std::ofstream output("foo.txt");

    // Write data.
    output << prepared_data;
}  // The ofstream will go out of scope here.
   // Its destructor will take care of closing the file properly.
```

Calling `close()` explicitly is only necessary if you want to reuse the same `fstream` object later, but don't want to keep the file open in between:

```
// Open the file "foo.txt" for the first time.
std::ofstream output("foo.txt");

// Get some data to write from somewhere.
std::string const prepared_data = prepare_data();
```

```
// Write data to the file "foo.txt".
output << prepared_data;

// Close the file "foo.txt".
output.close();

// Preparing data might take a long time. Therefore, we don't open the output file stream
// before we actually can write some data to it.
std::string const more_prepared_data = prepare_complex_data();

// Open the file "foo.txt" for the second time once we are ready for writing.
output.open("foo.txt");

// Write the data to the file "foo.txt".
output << more_prepared_data;

// Close the file "foo.txt" once again.
output.close();
```

Section 12.12: Reading a `struct` from a formatted text file

Version ≥ C++11

```
struct info_type
{
    std::string name;
    int age;
    float height;

    // we define an overload of operator>> as a friend function which
    // gives in privileged access to private data members
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // skip whitespace
        is >> std::ws;
        std::getline(is, info.name);
        is >> info.age;
        is >> info.height;
        return is;
    }
};

void func4()
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info;) // keep reading until we run out
    {
        // we only get here if the read succeeded
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << "  name: " << info.name << '\n';
        std::cout << "   age: " << info.age << " years" << '\n';
        std::cout << "height: " << info.height << "lbs" << '\n';
        std::cout << '\n';
    }
}
```

file4.txt

```
Wogger Wabbit
2
6.2
Bilbo Baggins
111
81.3
Mary Poppins
29
154.8
```

Output:

```
name: Wogger Wabbit
 age: 2 years
height: 6.2lbs

name: Bilbo Baggins
 age: 111 years
height: 81.3lbs

name: Mary Poppins
 age: 29 years
height: 154.8lbs
```

Chapter 13: C++ Streams

Section 13.1: String streams

`std::ostringstream` is a class whose objects look like an output stream (that is, you can write to them via `operator<<`), but actually store the writing results, and provide them in the form of a stream.

Consider the following short code:

```
#include <sstream>
#include <string>


using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

The line

```
ostringstream ss;
```

creates such an object. This object is first manipulated like a regular stream:

```
ss << "the answer to everything is " << 42;
```

Following that, though, the resulting stream can be obtained like this:

```
const string result = ss.str();
```

(the string `result` will be equal to `"the answer to everything is 42"`).

This is mainly useful when we have a class for which stream serialization has been defined, and for which we want a string form. For example, suppose we have some class

```
class foo
{
    // All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

To get the string representation of a `foo` object,

```
foo f;
```

we could use

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

Then `result` contains the string representation of the `foo` object.

Section 13.2: Printing collections with iostream

Basic printing

`std::ostream_iterator` allows to print contents of an STL container to any output stream without explicit loops. The second argument of `std::ostream_iterator` constructor sets the delimiter. For example, the following code:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

will print

```
1 ! 2 ! 3 ! 4 !
```

Implicit type cast

`std::ostream_iterator` allows to cast container's content type implicitly. For example, let's tune `std::cout` to print floating-point values with 3 digits after decimal point:

```
std::cout << std::setprecision(3);
std::fixed(std::cout);
```

and instantiate `std::ostream_iterator` with `float`, while the contained values remain `int`:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

so the code above yields

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

despite `std::vector` holds `int`s.

Generation and transformation

`std::generate`, `std::generate_n` and `std::transform` functions provide a very powerful tool for on-the-fly data manipulation. For example, having a vector:

```
std::vector<int> v = {1,2,3,4,8,16};
```

we can easily print boolean value of "x is even" statement for each element:

```
std::boolalpha(std::cout); // print booleans alphabetically
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),
[](int val) {
    return (val % 2) == 0;
});
```

or print the squared element:

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Printing N space-delimited random numbers:

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

Arrays

As in the section about reading text files, almost all these considerations may be applied to native arrays. For example, let's print squared values from a native array:

```
int v[] = {1,2,3,4,8,16};
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Chapter 14: Stream manipulators

Manipulators are special helper functions that help controlling input and output streams using `operator >>` or `operator <<`.

They all can be included by `#include <iomanip>`.

Section 14.1: Stream manipulators

`std::boolalpha` and `std::noboolalpha` - switch between textual and numeric representation of booleans.

```
std::cout << std::boolalpha << 1;
// Output: true

std::cout << std::noboolalpha << false;
// Output: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "Value \"" << std::boolalpha << boolValue
          << "\" was parsed as " << std::noboolalpha << boolValue;
// Input: true
// Output: Value "true" was parsed as 0
```

`std::showbase` and `std::noshowbase` - control whether prefix indicating numeric base is used.

`std::dec` (decimal), `std::hex` (hexadecimal) and `std::oct` (octal) - are used for changing base for integers.

```
#include <sstream>

std::cout << std::dec << 29 << ' - '
          << std::hex << 29 << ' - '
          << std::showbase << std::oct << 29 << ' - '
          << std::noshowbase << 29  '\n';
int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// Output: 22 - 1D - 35 - 035
// 59
```

Default values are `std::ios_base::noshowbase` and `std::ios_base::dec`.

If you want to see more about `std::istringstream` check out the <sstream> header.

`std::uppercase` and `std::nouppercase` - control whether uppercase characters are used in floating-point and hexadecimal integer output. Have no effect on input streams.

```
std::cout << std::hex << std::showbase
              << "0x2a with nouppercase: " << std::nouppercase << 0x2a << '\n'
              << "1e-10 with uppercase: " << std::uppercase << 1e-10 << '\n'
}
// Output: 0x2a with nouppercase: 0x2a
// 1e-10 with uppercase: 1E-10
```

Default is `std::nouppercase`.

`std::setw(n)` - changes the width of the next input/output field to exactly `n`.

The width property `n` is resetting to `0` when some functions are called (full list is here).

```
std::cout << "no setw:" << 51 << '\n'
          << "setw(7): " << std::setw(7) << 51 << '\n'
          << "setw(7), more output: " << 13
          << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << '\n';

char* input = "Hello, world!";
char arr[10];
std::cin >> std::setw(6) >> arr;
std::cout << "Input from \"Hello, world!\" with setw(6) gave \"" << arr << "\"\n";

// Output: 51
// setw(7):      51
// setw(7), more output: 13*****67 94

// Input: Hello, world!
// Output: Input from "Hello, world!" with setw(6) gave "Hello"
```

Default is `std::setw(0)`.

`std::left`, `std::right` and `std::internal` - modify the default position of the fill characters by setting `std::ios_base::adjustfield` to `std::ios_base::left`, `std::ios_base::right` and `std::ios_base::internal` correspondingly. `std::left` and `std::right` apply to any output, `std::internal` - for integer, floating-point and monetary output. Have no effect on input streams.

```
#include <locale>
...

std::cout.imbue(std::locale("en_US.utf8"));

std::cout << std::left << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, false) << '\n';
// Output:
// flt: -9.87**********
// hex: 41*************
//   $: $3.67**********
// usd: USD *3.67******
// usd: $3.67

std::cout << std::internal << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
```

```
// Output:
// flt: -**********9.87
// hex: *************41
//   $: $3.67**********
// usd: USD *******3.67
// usd: USD        3.67

std::cout << std::right << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: **********-9.87
// hex: *************41
//   $: **********$3.67
// usd: ******USD *3.67
// usd:       USD  3.67
```

Default is `std::left`.

`std::fixed`, `std::scientific`, `std::hexfloat` [C++11] and `std::defaultfloat` [C++11] - change formatting for floating-point input/output.

`std::fixed` sets the `std::ios_base::floatfield` to `std::ios_base::fixed`,
`std::scientific` - to `std::ios_base::scientific`,
`std::hexfloat` - to `std::ios_base::fixed | std::ios_base::scientific` and
`std::defaultfloat` - to `std::ios_base::fmtflags(0)`.

`fmtflags`

```
#include <sstream>
...

std::cout << '\n'
          << "The number 0.07 in fixed:      " << std::fixed << 0.01 << '\n'
          << "The number 0.07 in scientific: " << std::scientific << 0.01 << '\n'
          << "The number 0.07 in hexfloat:   " << std::hexfloat << 0.01 << '\n'
          << "The number 0.07 in default:    " << std::defaultfloat << 0.01 << '\n';

double f;
std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL);
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';

// Output:
// The number 0.01 in fixed:      0.070000
// The number 0.01 in scientific: 7.000000e-02
// The number 0.01 in hexfloat:   0x1.1eb851eb851ecp-4
// The number 0.01 in default:    0.07
// Parsing 0x1P-1022 as hex gives 2.22507e-308
```

Default is `std::ios_base::fmtflags(0)`.

There is a **bug** on some compilers which causes

```
double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
// Output: Parsing 0x1P-1022 as hex gives 0
```

std::showpoint and std::noshowpoint - control whether decimal point is always included in floating-point representation. Have no effect on input streams.

```
std::cout << "7.0 with showpoint: " << std::showpoint << 7.0 << '\n'
          << "7.0 with noshowpoint: " << std::noshowpoint << 7.0 << '\n';
// Output: 1.0 with showpoint: 7.00000
// 1.0 with noshowpoint: 7
```

Default is `std::showpoint`.

std::showpos and std::noshowpos - control displaying of the `+` sign in *non-negative* output. Have no effect on input streams.

```
std::cout << "With showpos: " << std::showpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n'
          << "Without showpos: " << std::noshowpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n';
// Output: With showpos: +0 -2.718 +17
// Without showpos: 0 -2.718 17
```

Default if `std::noshowpos`.

std::unitbuf, std::nounitbuf - control flushing output stream after every operation. Have no effect on input stream. `std::unitbuf` causes flushing.

std::setbase(base) - sets the numeric base of the stream.

`std::setbase(8)` equals to setting `std::ios_base::basefield` to `std::ios_base::oct`,
`std::setbase(16)` - to `std::ios_base::hex`,
`std::setbase(10)` - to `std::ios_base::dec`.

If `base` is other then `8`, `10` or `16` then `std::ios_base::basefield` is setting to `std::ios_base::fmtflags(0)`. It means decimal output and prefix-dependent input.

As default `std::ios_base::basefield` is `std::ios_base::dec` then by default `std::setbase(10)`.

std::setprecision(n) - changes floating-point precision.

```
#include <cmath>
#include <limits>
...

typedef std::numeric_limits<long double> ld;
```

```
const long double pi = std::acos(-1.L);

std::cout << '\n'
          << "default precision (6):   pi: " << pi << '\n'
          << "                       10pi: " << 10 * pi << '\n'
          << "std::setprecision(4):  10pi: " << std::setprecision(4) << 10 * pi << '\n'
          << "                    10000pi: " << 10000 * pi << '\n'
          << "std::fixed:         10000pi: " << std::fixed << 10000 * pi << std::defaultfloat <<
'\n'
          << "std::setprecision(10):   pi: " << std::setprecision(10) << pi << '\n'
          << "max-1 radix precicion:   pi: " << std::setprecision(ld::digits - 1) << pi << '\n'
          << "max+1 radix precision:   pi: " << std::setprecision(ld::digits + 1) << pi << '\n'
          << "significant digits prec: pi: " << std::setprecision(ld::digits10) << pi << '\n';

// Output:
// default precision (6):   pi: 3.14159
//                        10pi: 31.4159
// std::setprecision(4):  10pi: 31.42
//                     10000pi: 3.142e+04
// std::fixed:         10000pi: 31415.9265
// std::setprecision(10):   pi: 3.141592654
// max-1 radix precicion:   pi: 3.14159265358979323851280895940618620443274267017841339111328125
// max+1 radix precision:   pi: 3.14159265358979323851280895940618620443274267017841339111328125
// significant digits prec: pi: 3.14159265358979324
```

Default is `std::setprecision(6)`.

`std::setiosflags(mask)` and `std::resetiosflags(mask)` - set and clear flags specified in `mask` of `std::ios_base::fmtflags` type.

```
#include <sstream>
...

std::istringstream in("10 010 10 010 10 010");
int num1, num2;

in >> std::oct >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::oct gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   8 8

in >> std::dec >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::dec gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2;
std::cout << "Parsing \"10 010\" with autodetect gives: " << num1 << ' ' << num2 << '\n';
// Parsing "10 010" with autodetect gives: 10 8

std::cout << std::setiosflags(std::ios_base::hex |
                              std::ios_base::uppercase |
                              std::ios_base::showbase) << 42 << '\n';
// Output: OX2A
```

`std::skipws` and `std::noskipws` - control skipping of leading whitespace by the formatted input functions. Have no effect on output streams.

```
#include <sstream>
```

```
...

char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "Default  behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';

std::istringstream("a b c") >> std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';
// Output: Default  behavior:  c1 = a  c2 = b  c3 = c
// noskipws behavior:  c1 = a  c2 =    c3 = b
```

Default is `std::ios_base::skipws`.

`std::quoted(s[, delim[, escape]])` [C++14] - inserts or extracts quoted strings with embedded spaces.

`s` - the string to insert or extract.
`delim` - the character to use as the delimiter, `"` by default.
`escape` - the character to use as the escape character, `\` by default.

```
#include <sstream>
...

std::stringstream ss;
std::string in = "String with spaces, and embedded \"quotes\" too";
std::string out;

ss << std::quoted(in);
std::cout << "read in     [" << in << "]\n"
          << "stored as   [" << ss.str() << "]\n";

ss >> std::quoted(out);
std::cout << "written out [" << out << "]\n";
// Output:
// read in     [String with spaces, and embedded "quotes" too]
// stored as   ["String with spaces, and embedded \"quotes\" too"]
// written out [String with spaces, and embedded "quotes" too]
```

For more information see the link above.

Section 14.2: Output stream manipulators

`std::ends` - inserts a null character `'\0'` to output stream. More formally this manipulator's declaration looks like

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

and this manipulator places character by calling `os.put(charT())` when used in an expression
`os << std::ends;`

`std::endl` and `std::flush` both flush output stream `out` by calling `out.flush()`. It causes immediately producing output. But `std::endl` inserts end of line `'\n'` symbol before flushing.

```
std::cout << "First line." << std::endl << "Second line. " << std::flush
          << "Still second line.";
```

```
// Output: First line.
// Second line. Still second line.
```

std::setfill(c) - changes the fill character to c. Often used with std::setw.

```
std::cout << "\nDefault fill: " << std::setw(10) << 79 << '\n'
          << "setfill('#'): " << std::setfill('#')
          << std::setw(10) << 42 << '\n';
// Output:
// Default fill:         79
// setfill('#'): ########79
```

std::put_money(mon[, intl]) [C++11]. In an expression out << std::put_money(mon, intl), converts the monetary value mon (of long double or std::basic_string type) to its character representation as specified by the std::money_put facet of the locale currently imbued in out. Use international currency strings if intl is true, use currency symbols otherwise.

```
long double money = 123.45;
// or std::string money = "123.45";

std::cout.imbue(std::locale("en_US.utf8"));
std::cout << std::showbase << "en_US: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: en_US: $1.23 or USD  1.23

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "ru_RU: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ru_RU: 1.23 руб or 1.23 RUB

std::cout.imbue(std::locale("ja_JP.utf8"));
std::cout << "ja_JP: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ja_JP: ￥123 or JPY  123
```

std::put_time(tmb, fmt) [C++11] - formats and outputs a date/time value to std::tm according to the specified format fmt.

tmb - pointer to the calendar time structure const std::tm* as obtained from localtime() or gmtime().
fmt - pointer to a null-terminated string const CharT* specifying the format of conversion.

```
#include <ctime>
...

std::time_t t = std::time(nullptr);
std::tm tm = *std::localtime(&t);

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "\nru_RU: " << std::put_time(&tm, "%c %Z") << '\n';
// Possible output:
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

For more information see the link above.

Section 14.3: Input stream manipulators

`std::ws` - consumes leading whitespaces in input stream. It different from `std::skipws`.

```
#include <sstream>
...

std::string str;
std::istringstream("  \v\n\r\t    Wow!There   is no whitespaces!") >> std::ws >> str;
std::cout << str;
// Output: Wow!There   is no whitespaces!
```

`std::get_money(mon[, intl])` [C++11]. In an expression `in >> std::get_money(mon, intl)` parses the character input as a monetary value, as specified by the `std::money_get` facet of the locale currently imbued in `in`, and stores the value in `mon` (of `long double` or `std::basic_string` type). Manipulator expects *required* international currency strings if `intl` is `true`, expects *optional* currency symbols otherwise.

```
#include <sstream>
#include <locale>
...

std::istringstream in("$1,234.56 2.22 USD  3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " parsed as: "
              << v1 << ", " << v2 << ", " << v3 << '\n';
}
// Output:
// "$1,234.56 2.22 USD  3.33" parsed as: 123456, 222, 333
```

`std::get_time(tmb, fmt)` [C++11] - parses a date/time value stored in `tmb` of specified format `fmt`.

`tmb` - valid pointer to the `const std::tm*` object where the result will be stored.
`fmt` - pointer to a null-terminated string `const CharT*` specifying the conversion format.

```
#include <sstream>
#include <locale>
...

std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "Parse failed\n";
}
else {
    std::cout << std::put_time(&t, "%c") << '\n';
}
// Possible output:
// Sun Feb 18 23:12:34 2011
```

For more information see the link above.

Chapter 15: Flow Control

Section 15.1: case

Introduces a case label of a switch statement. The operand must be a constant expression and match the switch condition in type. When the switch statement is executed, it will jump to the case label with operand equal to the condition, if any.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

Section 15.2: switch

According to the C++ standard,

> The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

The keyword `switch` is followed by a parenthesized condition and a block, which may contain `case` labels and an optional `default` label. When the switch statement is executed, control will be transferred either to a `case` label with a value matching that of the condition, if any, or to the `default` label, if any.

The condition must be an expression or a declaration, which has either integer or enumeration type, or a class type with a conversion function to integer or enumeration type.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

Section 15.3: catch

The `catch` keyword introduces an exception handler, that is, a block into which control will be transferred when an exception of compatible type is thrown. The `catch` keyword is followed by a parenthesized *exception declaration*, which is similar in form to a function parameter declaration: the parameter name may be omitted, and the ellipsis `...` is allowed, which matches any type. The exception handler will only handle the exception if its declaration is compatible with the type of the exception. For more details, see catching exceptions.

```
try {
    std::vector<int> v(N);
    // do something
} catch (const std::bad_alloc&) {
    std::cout << "failed to allocate memory for vector!" << std::endl;
} catch (const std::runtime_error& e) {
    std::cout << "runtime error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "unexpected exception!" << std::endl;
    throw;
}
```

Section 15.4: throw

1. When `throw` occurs in an expression with an operand, its effect is to throw an exception, which is a copy of the operand.

    ```
    void print_asterisks(int count) {
        if (count < 0) {
            throw std::invalid_argument("count cannot be negative!");
        }
        while (count--) { putchar('*'); }
    }
    ```

2. When `throw` occurs in an expression without an operand, its effect is to rethrow the current exception. If there is no current exception, `std::terminate` is called.

    ```
    try {
        // something risky
    } catch (const std::bad_alloc&) {
        std::cerr << "out of memory" << std::endl;
    } catch (...) {
        std::cerr << "unexpected exception" << std::endl;
        // hope the caller knows how to handle this exception
        throw;
    }
    ```

3. When `throw` occurs in a function declarator, it introduces a dynamic exception specification, which lists the types of exceptions that the function is allowed to propagate.

    ```
    // this function might propagate a std::runtime_error,
    // but not, say, a std::logic_error
    void risky() throw(std::runtime_error);
    // this function can't propagate any exceptions
    void safe() throw();
    ```

    Dynamic exception specifications are deprecated as of C++11.

Note that the first two uses of `throw` listed above constitute expressions rather than statements. (The type of a throw expression is `void`.) This makes it possible to nest them within expressions, like so:

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

Section 15.5: default

In a switch statement, introduces a label that will be jumped to if the condition's value is not equal to any of the case labels' values.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

Version ≥ C++11

Defines a default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move assignment operator to have its default behaviour.

```
class Base {
    // ...
    // we want to be able to delete derived classes through Base*,
    // but have the usual behaviour for Base's destructor.
    virtual ~Base() = default;
};
```

Section 15.6: try

The keyword `try` is followed by a block, or by a constructor initializer list and then a block (see here). The try block is followed by one or more catch blocks. If an exception propagates out of the try block, each of the corresponding catch blocks after the try block has the opportunity to handle the exception, if the types match.

```
std::vector<int> v(N);     // if an exception is thrown here,
                           // it will not be caught by the following catch block
try {
    std::vector<int> v(N); // if an exception is thrown here,
                           // it will be caught by the following catch block
    // do something with v
} catch (const std::bad_alloc&) {
    // handle bad_alloc exceptions from the try block
}
```

Section 15.7: if

Introduces an if statement. The keyword `if` must be followed by a parenthesized condition, which can be either an expression or a declaration. If the condition is truthy, the substatement after the condition will be executed.

```
int x;
```

```
std::cout << "Please enter a positive number." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "You didn't enter a positive number!" << std::endl;
    abort();
}
```

Section 15.8: else

The first substatement of an if statement may be followed by the keyword `else`. The substatement after the `else` keyword will be executed when the condition is falsey (that is, when the first substatement is not executed).

```
int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}
```

Section 15.9: Conditional Structures: if, if..else

if and else:

it used to check whether the given expression returns true or false and acts as such:

```
if (condition) statement
```

the condition can be any valid C++ expression that returns something that be checked against truth/falsehood for example:

```
if (true) { /* code here */ }  // evaluate that true is true and execute the code in the brackets
if (false) { /* code here */ } // always skip the code since false is always false
```

the condition can be anything, a function, a variable, or a comparison for example

```
if(istrue()) { } // evaluate the function, if it returns true, the if will execute the code
if(isTrue(var)) { } //evaluate the return of the function after passing the argument var
if(a == b) { } // this will evaluate the return of the experssion (a==b) which will be true if
equal and false if unequal
if(a) { } //if a is a boolean type, it will evaluate for its value, if it's an integer, any non
zero value will be true,
```

if we want to check for a multiple expressions we can do it in two ways :

using binary operators :

```
if (a && b) { } // will be true only if both a and b are true (binary operators are outside the
scope here
if (a || b ) { } //true if a or b is true
```

using if/ifelse/else:

for a simple switch either if or else

```
if (a== "test") {
    //will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

for multiple choices :

```
if (a=='a') {
// if a is a char valued 'a'
} else if (a=='b') {
// if a is a char valued 'b'
} else if (a=='c') {
// if a is a char valued 'c'
} else {
//if a is none of the above
}
```

however it must be noted that you should use '**switch**' instead if your code checks for the same variable's value

Section 15.10: goto

Jumps to a labelled statement, which must be located in the current function.

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // we can't continue, but must do cleanup still
        goto end;
    }
    // ...
    result = true;
  end:
    release_widget(widget);
    return result;
}
```

Section 15.11: Jump statements : break, continue, goto, exit

The break instruction:

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end

The syntax is

```
break;
```

Example: we often use `break` in `switch` cases,ie once a case i switch is satisfied then the code block of that condition is executed .

```
switch(conditon){
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

in this case if case 1 is satisfied then block 1 is executed , what we really want is only the block1 to be processed but instead once the block1 is processed remaining blocks,block2,block3 and blockdefault are also processed even though only case 1 was satified.To avoid this we use break at the end of each block like :

```
switch(condition){
case 1: block1;
        break;
case 2: block2;
        break;
case 3: block3;
        break;
default: blockdefault;
        break;
}
```

so only one block is processed and the control moves out of the switch loop.

break can also be used in other conditional and non conditional loops like `if`,`while`,`for` etc;

example:

```
if(condition1){
   ....
   if(condition2){
    .......
    break;
    }
 ...
}
```

The continue instruction:

The continue instruction causes the program to skip the rest of the loop in the present iteration as if the end of the statement block would have been reached, causing it to jump to the following iteration.

The syntax is

```
continue;
```

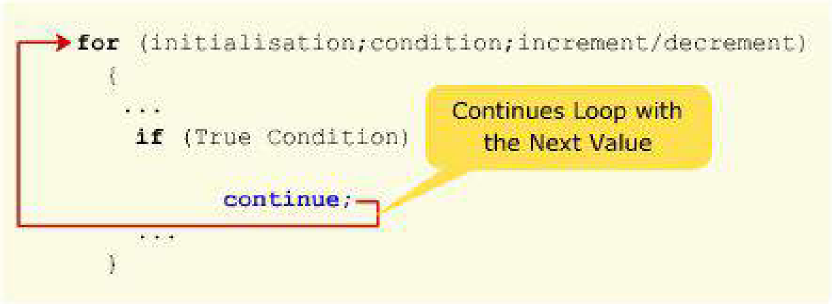
Example consider the following :

```
for(int i=0;i<10;i++){
if(i%2==0)
continue;
cout<<"\n @"<<i;
}
```

which produces the output:

```
 @1
 @3
 @5
 @7
 @9
```

i this code whenever the condition `i%2==0` is satisfied `continue` is processed,this causes the compiler to skip all the remaining code(printing @ and i) and increment/decrement statement of the loop gets executed.



```
for (initialisation;condition;increment/decrement)
  {
   ...
    if (True Condition)

          continue;
    ...
  }
```

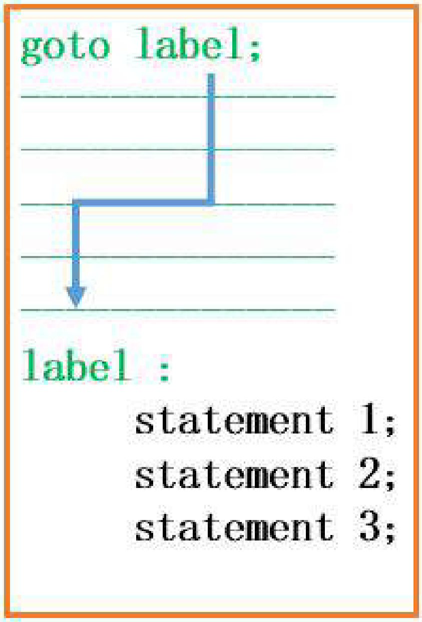
The goto instruction:

It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation. The destination point is identified by a label, which is then used as an argument for the goto instruction. A label is made of a valid identifier followed by a colon (:)

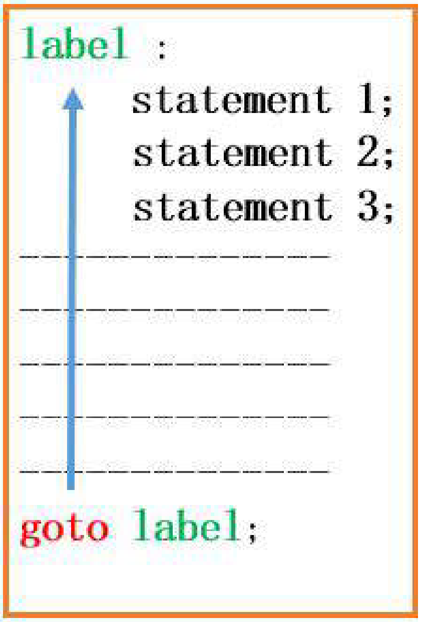
The syntax is

```
goto label;
..
.
label: statement;
```

Note: *Use of goto statement is highly discouraged because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify.*



Forward Reference



Backward Reference

Example :

```
int num = 1;
STEP:
do{

    if( num%2==0 )
    {
        num = num + 1;
        goto STEP;
     }

   cout << "value of num : " << num << endl;
   num = num + 1;
 }while( num < 10 );
```

output :

```
value of num : 1
value of num : 3
value of num : 5
value of num : 7
value of num : 9
```

whenever the condition `num%2==0` is satisfied the goto sends the execution control to the beginning of the `do-while` loop.

The exit function:

`exit` is a function defined in `cstdlib`. The purpose of `exit` is to terminate the running program with an specific exit code. Its prototype is:

```
void exit (int exit code);
```

`cstdlib` defines the standard exit codes `EXIT_SUCCESS` and `EXIT_FAILURE`.

Section 15.12: return

Returns control from a function to its caller.

If `return` has an operand, the operand is converted to the function's return type, and the converted value is returned to the caller.

```
int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3
```

If `return` does not have an operand, the function must have `void` return type. As a special case, a `void`-returning function can also return an expression if the expression has type `void`.

```
void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // calls f, then returns
    return g(); // ill-formed
}
```

When `main` returns, `std::exit` is implicitly called with the return value, and the value is thus returned to the execution environment. (However, returning from `main` destroys automatic local variables, while calling `std::exit` directly does not.)

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}
```

Chapter 16: Metaprogramming

In C++ Metaprogramming refers to the use of macros or templates to generate code at compile-time.

In general, macros are frowned upon in this role and templates are preferred, although they are not as generic.

Template metaprogramming often makes use of compile-time computations, whether via templates or `constexpr` functions, to achieve its goals of generating code, however compile-time computations are not metaprogramming per se.

Section 16.1: Calculating Factorials

Factorials can be computed at compile-time using template metaprogramming techniques.

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

`factorial` is a struct, but in template metaprogramming it is treated as a template metafunction. By convention, template metafunctions are evaluated by checking a particular member, either `::type` for metafunctions that result in types, or `::value` for metafunctions that generate values.

In the above code, we evaluate the `factorial` metafunction by instantiating the template with the parameters we want to pass, and using `::value` to get the result of the evaluation.

The metafunction itself relies on recursively instantiating the same metafunction with smaller values. The `factorial<0>` specialization represents the terminating condition. Template metaprogramming has most of the restrictions of a functional programming language, so recursion is the primary "looping" construct.

Since template metafunctions execute at compile time, their results can be used in contexts that require compile-time values. For example:

```
int my_array[factorial<5>::value];
```

Automatic arrays must have a compile-time defined size. And the result of a metafunction is a compile-time constant, so it can be used here.

Limitation: Most of the compilers won't allow recursion depth beyond a limit. For example, `g++` compiler by default limits recursion depeth to 256 levels. In case of `g++`, programmer can set recursion depth using `-ftemplate-depth-X` option.

Version ≥ C++11

Since C++11, the `std::integral_constant` template can be used for this kind of template computation:

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial :
  std::integral_constant<long long, n * factorial<n - 1>::value> {};

template<>
struct factorial<0> :
  std::integral_constant<long long, 1> {};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

Additionally, `constexpr` functions become a cleaner alternative.

```
#include <iostream>

constexpr long long factorial(long long n)
{
  return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
  char test[factorial(3)];
  std::cout << factorial(7) << '\n';
}
```

The body of `factorial()` is written as a single statement because in C++11 `constexpr` functions can only use a quite limited subset of the language.

Version ≥ C++14

Since C++14, many restrictions for `constexpr` functions have been dropped and they can now be written much more conveniently:

```
constexpr long long factorial(long long n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Or even:

```
constexpr long long factorial(int n)
{
  long long result = 1;
  for (int i = 1; i <= n; ++i) {
    result *= i;
  }
  return result;
}
```

Version ≥ C++17

Since c++17 one can use fold expression to calculate factorial:

```
#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;

template <class T, T N, T... Is>
struct factorial<T,N,std::index_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
   std::cout << factorial<int, 5>::value << std::endl;
}
```

Section 16.2: Iterating over a parameter pack

Often, we need to perform an operation over every element in a variadic template parameter pack. There are many ways to do this, and the solutions get easier to read and write with C++17. Suppose we simply want to print every element in a pack. The simplest solution is to recurse:

Version ≥ C++11

```
void print_all(std::ostream& os) {
    // base case
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}
```

We could instead use the expander trick, to perform all the streaming in a single function. This has the advantage of not needing a second overload, but has the disadvantage of less than stellar readability:

Version ≥ C++11

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

For an explanation of how this works, see T.C's excellent answer.

Version ≥ C++17

With C++17, we get two powerful new tools in our arsenal for solving this problem. The first is a fold-expression:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

And the second is `if constexpr`, which allows us to write our original recursive solution in a single function:

```
template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    if constexpr (sizeof...(rest) > 0) {
        // this line will only be instantiated if there are further
        // arguments. if rest... is empty, there will be no call to
        // print_all(os).
        print_all(os, rest...);
    }
}
```

Section 16.3: Iterating with std::integer_sequence

Since C++14, the standard provides the class template

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

and a generating metafunction for it:

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

While this comes standard in C++14, this can be implemented using C++11 tools.

We can use this tool to call a function with a `std::tuple` of arguments (standardized in C++17 as `std::apply`):

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...> ) {
        return std::forward<F>(f)(std::get<Is>(std::forward<Tuple>(tpl))...);
    }
}

template <class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& tpl) {
    return detail::apply_impl(std::forward<F>(f),
        std::forward<Tuple>(tpl),
        std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
}


// this will print 3
int f(int, char, double);
```

```
auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // calls f(42, 'x', 3.14)
```

Section 16.4: Tag Dispatching

A simple way of selecting between functions at compile time is to dispatch a function to an overloaded pair of functions that take a tag as one (usually the last) argument. For example, to implement `std::advance()`, we can dispatch on the iterator category:

```
namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class BidirIter, class Distance>
    void advance(BidirIter& it, Distance n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class InputIter, class Distance>
    void advance(InputIter& it, Distance n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }
}

template <class Iter, class Distance>
void advance(Iter& it, Distance n) {
    details::advance(it, n,
            typename std::iterator_traits<Iter>::iterator_category{} );
}
```

The `std::XY_iterator_tag` arguments of the overloaded `details::advance` functions are unused function parameters. The actual implementation does not matter (actually it is completely empty). Their only purpose is to allow the compiler to select an overload based on which tag class `details::advance` is called with.

In this example, `advance` uses the `iterator_traits<T>::iterator_category` metafunction which returns one of the `iterator_tag` classes, depending on the actual type of `Iter`. A default-constructed object of the `iterator_category<Iter>::type` then lets the compiler select one of the different overloads of `details::advance`. (This function parameter is likely to be completely optimized away, as it is a default-constructed object of an empty `struct` and never used.)

Tag dispatching can give you code that's much easier to read than the equivalents using SFINAE and `enable_if`.

Note: while C++17's `if constexpr` may simplify the implementation of `advance` in particular, it is not suitable for open implementations unlike tag dispatching.

Section 16.5: Detect Whether Expression is Valid

It is possible to detect whether an operator or function can be called on a type. To test if a class has an overload of std::hash, one can do this:

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type and std::true_type
#include <utility> // for std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>()(std::declval<T>()), void())>
    : std::true_type
{};
```

Version ≥ C++17

Since C++17, `std::void_t` can be used to simplify this type of construct

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type, std::true_type, std::void_t
#include <utility> // for std::declval

template<class, class = std::void_t<> >
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t< decltype(std::hash<T>()(std::declval<T>())) > >
    : std::true_type
{};
```

where `std::void_t` is defined as:

```
template< class... > using void_t = void;
```

For detecting if an operator, such as `operator<` is defined, the syntax is almost the same:

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};
```

These can be used to use a `std::unordered_map<T>` if `T` has an overload for `std::hash`, but otherwise attempt to use a `std::map<T>`:

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K,V>>;
```

Section 16.6: If-then-else

Version ≥ C++11

The type std::conditional in the standard library header **<type_traits>** can select one type or the other, based on a compile-time boolean value:

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

This struct contains a pointer to T if T is larger than the size of a pointer, or T itself if it is smaller or equal to a pointer's size. Therefore sizeof(ValueOrPointer) will always be <= sizeof(void*).

Section 16.7: Manual distinction of types when given any type T

When implementing SFINAE using std::enable_if, it is often useful to have access to helper templates that determines if a given type T matches a set of criteria.

To help us with that, the standard already provides two types analog to true and false which are std::true_type and std::false_type.

The following example show how to detect if a type T is a pointer or not, the is_pointer template mimic the behavior of the standard std::is_pointer helper:

```
template <typename T>
struct is_pointer_: std::false_type {};

template <typename T>
struct is_pointer_<T*>: std::true_type {};

template <typename T>
struct is_pointer: is_pointer_<typename std::remove_cv<T>::type> { }
```

There are three steps in the above code (sometimes you only need two):

1. The first declaration of is_pointer_ is the *default case*, and inherits from std::false_type. The *default* case should always inherit from std::false_type since it is analogous to a "false condition".
2. The second declaration specialize the is_pointer_ template for pointer T* without caring about what T is really. This version inherits from std::true_type.
3. The third declaration (the real one) simply remove any unnecessary information from T (in this case we remove const and volatile qualifiers) and then fall backs to one of the two previous declarations.

Since is_pointer<T> is a class, to access its value you need to either:

- Use ::value, e.g. is_pointer<int>::value – value is a static class member of type bool inherited from std::true_type or std::false_type;
- Construct an object of this type, e.g. is_pointer<int>{} – This works because std::is_pointer inherits its default constructor from std::true_type or std::false_type (which have constexpr constructors) and both std::true_type and std::false_type have constexpr conversion operators to bool.

It is a good habit to provides "helper helper templates" that let you directly access the value:

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

Version ≥ C++17

In C++17 and above, most helper templates already provide a `_v` version, e.g.:

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

Section 16.8: Calculating power with C++11 (and higher)

With C++11 and higher calculations at compile time can be much easier. For example calculating the power of a given number at compile time will be following:

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

Keyword `constexpr` is responsible for calculating function in compilation time, then and only then, when all the requirements for this will be met (see more at constexpr keyword reference) for example all the arguments must be known at compile time.

Note: In C++11 `constexpr` function must compose only from one return statement.

Advantages: Comparing this to the standard way of compile time calculation, this method is also useful for runtime calculations. It means, that if the arguments of the function are not known at the compilation time (e.g. value and power are given as input via user), then function is run in a compilation time, so there's no need to duplicate a code (as we would be forced in older standards of C++).

E.g.

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // computes at compile time,
                               // as both arguments are known at compilation time
                               // and used for a constant expression.
    int value;
    std::cin >> value;
    int runtimeCalculated = calculatePower(value, 3);  // runtime calculated,
                                    // because value is known only at runtime.
}
```

Version ≥ C++17

Another way to calculate power at compile time can make use of fold expression as follows:

```
#include <iostream>
#include <utility>

template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};
```

```
int main() {
   std::cout << power<int, 4, 2>::value << std::endl;
}
```

Section 16.9: Generic Min/Max with variable argument count

Version > C++11

It's possible to write a generic function (for example `min`) which accepts various numerical types and arbitrary argument count by template meta-programming. This function declares a `min` for two arguments and recursively for more.

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
auto min(const T1 &a, const T2 &b, const Args& ... args)
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

Chapter 17: const keyword

Section 17.1: Avoiding duplication of code in const and non-const getter methods

In C++ methods that differs only by `const` qualifier can be overloaded. Sometimes there may be a need of two versions of getter that return a reference to some member.

Let `Foo` be a class, that has two methods that perform identical operations and returns a reference to an object of type `Bar`:

```
class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};
```

The only difference between them is that one method is non-const and return a non-const reference (that can be use to modify object) and the second is const and returns const reference.

To avoid the code duplication, there is a temptation to call one method from another. However, we can not call non-const method from the const one. But we can call const method from non-const one. That will require as to use 'const_cast' to remove the const qualifier.

The solution is:

```
struct Foo
{
    Bar& GetBar(/*arguments*/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/*arguments*/));
    }

    const Bar& GetBar(/*arguments*/) const
    {
        /* some calculations */
        return foo;
    }
};
```

In code above, we call const version of `GetBar` from the non-const `GetBar` by casting this to const type: `const_cast<const Foo*>(this)`. Since we call const method from non-const, the object itself is non-const, and casting away the const is allowed.

Examine the following more complete example:

```
#include <iostream>

class Student
{
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // non-const object
    Student a;
    // We can assign to the reference. Non-const version of GetScore is called
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // We still can call GetScore method of const object,
    // because we have overloaded const version of GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << '\n';
}
```

Section 17.2: Const member functions

Member functions of a class can be declared `const`, which tells the compiler and future readers that this function will not modify the object:

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

In a `const` member function, the `this` pointer is effectively a `const MyClass *` instead of a `MyClass *`. This means that you cannot change any member variables within the function; the compiler will emit a warning. So `setMyInt` could not be declared `const`.

You should almost always mark member functions as `const` when possible. Only `const` member functions can be called on a `const MyClass`.

`static` methods cannot be declared as `const`. This is because a static method belongs to a class and is not called on object; therefore it can never modify object's internal variables. So declaring `static` methods as `const` would be redundant.

Section 17.3: Const local variables

Declaration and usage.

```
// a is const int, so it can't be changed
const int a = 15;
a = 12;           // Error: can't assign new value to const variable
a += 1;           // Error: can't assign new value to const variable
```

Binding of references and pointers

```
int &b = a;       // Error: can't bind non-const reference to const variable
const int &c = a; // OK; c is a const reference

int *d = &a;      // Error: can't bind pointer-to-non-const to const variable
const int *e = &a // OK; e is a pointer-to-const

int f = 0;
e = &f;           // OK; e is a non-const pointer-to-const,
                  // which means that it can be rebound to new int* or const int*

*e = 1            // Error: e is a pointer-to-const which means that
                  // the value it points to can't be changed through dereferencing e

int *g = &f;
*g = 1;           // OK; this value still can be changed through dereferencing
                  // a pointer-not-to-const
```

Section 17.4: Const pointers

```
int a = 0, b = 2;

const int* pA = &a; // pointer-to-const. `a` can't be changed through this
int* const pB = &a; // const pointer. `a` can be changed, but this pointer can't.
const int* const pC = &a; // const pointer-to-const.

//Error: Cannot assign to a const reference
*pA = b;

pA = &b;

*pB = b;

//Error: Cannot assign to const pointer
pB = &b;

//Error: Cannot assign to a const reference
*pC = b;
```

```
//Error: Cannot assign to const pointer
pC = &b;
```

Chapter 18: mutable keyword

Section 18.1: mutable lambdas

By default, the implicit `operator()` of a lambda is `const`. This disallows performing non-`const` operations on the lambda. In order to allow modifying members, a lambda may be marked `mutable`, which makes the implicit `operator()` non-`const`:

```
int a = 0;

auto bad_counter = [a] {
    return a++;   // error: operator() is const
                  // cannot modify members
};

auto good_counter = [a]() mutable {
    return a++;  // OK
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

Section 18.2: non-static class member modifier

`mutable` modifier in this context is used to indicate that a data field of a const object may be modified without affecting the externally-visible state of the object.

If you are thinking about caching a result of expensive computation, you should probably use this keyword.

If you have a lock (for example, `std::unique_lock`) data field which is locked and unlocked inside a const method, this keyword is also what you could use.

You should not use this keyword to break logical const-ness of an object.

Example with caching:

```
class pi_calculator {
 public:
     double get_pi() const {
         if (pi_calculated) {
             return pi;
         } else {
             double new_pi = 0;
             for (int i = 0; i < 1000000000; ++i) {
                 // some calculation to refine new_pi
             }
             // note: if pi and pi_calculated were not mutable, we would get an error from a
compiler
             // because in a const method we can not change a non-mutable field
             pi = new_pi;
             pi_calculated = true;
             return pi;
         }
     }
 private:
     mutable bool pi_calculated = false;
```

```
    mutable double pi = 0;
};
```

Chapter 19: Friend keyword

Well-designed classes encapsulate their functionality, hiding their implementation while providing a clean, documented interface. This allows redesign or change so long as the interface is unchanged.

In a more complex scenario, multiple classes that rely on each others' implementation details may be required. Friend classes and functions allow these peers access to each others' details, without compromising the encapsulation and information hiding of the documented interface.

Section 19.1: Friend function

A class or a structure may declare any function it's friend. If a function is a friend of a class, it may access all it's protected and private members:

```
// Forward declaration of functions.
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // Declare one of the function as a friend.
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // Compilation error: private_value is private.
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // OK: friends may access private values.
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

Access modifiers do not alter friend semantics. Public, protected and private declarations of a friend are equivalent.

Friend declarations are not inherited. For example, if we subclass `PrivateHolder`:

```
class PrivateHolderDerived : public PrivateHolder {
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

and try to access it's members, we'll get the following:

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // OK.
    std::cout << pd.private_value << std::endl;
    // Compilation error: derived_private_value is private.
    std::cout << pd.derived_private_value << std::endl;
}
```

Note that `PrivateHolderDerived` member function cannot access `PrivateHolder::private_value`, while friend function can do it.

Section 19.2: Friend method

Methods may declared as friends as well as functions:

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: this method is declares as friend.
    std::cout << ph.private_value << std::endl;
}
```

Section 19.3: Friend class

A whole class may be declared as friend. Friend class declaration means that any member of the friend may access private and protected members of the declaring class:

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
}
```

Friend class declaration is not reflexive. If classes need private access in both directions, both of them need friend declarations.

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
private:
    int private_value = 0;
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    // Accesser is a friend of PrivateHolder
    friend class Accesser;
    void reverse_accesse() {
        // but PrivateHolder cannot access Accesser's members.
        Accesser a;
        std::cout << a.private_value;
    }
private:
    int private_value;
};
```

Chapter 20: Type Keywords

Section 20.1: class

1. Introduces the definition of a class type.

```
class foo {
    int x;
  public:
    int get_x();
    void set_x(int new_x);
};
```

2. Introduces an *elaborated type specifier,* which specifies that the following name is the name of a class type. If the class name has been declared already, it can be found even if hidden by another name. If the class name has not been declared already, it is forward-declared.

```
class foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
class baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
class foo {
    bar b;
    friend class baz; // elaborated type specifier refers to the class,
                      // not the function of the same name
  public:
    foo();
};
```

3. Introduces a type parameter in the declaration of a template.

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. In the declaration of a template template parameter, the keyword `class` precedes the name of the parameter. Since the argument for a template template parameter can only be a class template, the use of `class` here is redundant. However, the grammar of C++ requires it.

```
template <template <class T> class U>
//                           ^^^^^ "class" used in this sense here;
//                                 U is a template template parameter
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

5. Note that sense 2 and sense 3 may be combined in the same declaration. For example:

```
template <class T>
class foo {
};

foo<class bar> x; // <- bar does not have to have previously appeared.
```

Version ≥ C++11

6. In the declaration or definition of an enum, declares the enum to be a scoped enum.

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

Section 20.2: enum

1. Introduces the definition of an enumeration type.

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

Version ≥ C++11

In C++11, `enum` may optionally be followed by `class` or `struct` to define a scoped enum. Furthermore, both scoped and unscoped enums can have their underlying type explicitly specified by `: T` following the enum name, where `T` refers to an integer type.

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;

enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

Enumerators in normal `enum`s may also be preceded by the scope operator, although they are still considered to be in the scope the `enum` was defined in.

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

2. Introduces an *elaborated type specifier,* which specifies that the following name is the name of a previously declared enum type. (An elaborated type specifier cannot be used to forward-declare an enum type.) An

enum can be named in this way even if hidden by another name.

```
enum Foo { FOO };
void Foo() {}
Foo foo = FOO;      // ill-formed; Foo refers to the function
enum Foo foo = FOO; // ok; Foo refers to the enum type
```

Version ≥ C++11

3. Introduces an *opaque enum declaration,* which declares an enum without defining it. It can either redeclare a previously declared enum, or forward-declare an enum that has not been previously declared.

   An enum first declared as scoped cannot later be declared as unscoped, or *vice versa.* All declarations of an enum must agree in underlying type.

   When forward-declaring an unscoped enum, the underlying type must be explicitly specified, since it cannot be inferred until the values of the enumerators are known.

```
enum class Format; // underlying type is implicitly int
void f(Format f);
enum class Format {
    TEXT,
    PDF,
    OTHER,
};

enum Direction;    // ill-formed; must specify underlying type
```

Section 20.3: struct

Interchangeable with `class`, except for the following differences:

- If a class type is defined using the keyword `struct`, then the default accessibility of bases and members is `public` rather than `private`.
- `struct` cannot be used to declare a template type parameter or template template parameter; only `class` can.

Section 20.4: union

1. Introduces the definition of a union type.

```
// Example is from POSIX
union sigval {
    int     sival_int;
    void   *sival_ptr;
};
```

2. Introduces an *elaborated type specifier,* which specifies that the following name is the name of a union type. If the union name has been declared already, it can be found even if hidden by another name. If the union name has not been declared already, it is forward-declared.

```
union foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
union baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
union foo {
    long l;
    union baz* b; // elaborated type specifier refers to the class,
                  // not the function of the same name
};
```

Chapter 21: Basic Type Keywords

Section 21.1: char

An integer type which is "large enough to store any member of the implementation's basic character set". It is implementation-defined whether `char` is signed (and has a range of at least -127 to +127, inclusive) or unsigned (and has a range of at least 0 to 255, inclusive).

```
const char zero = '0';
const char one = zero + 1;
const char newline = '\n';
std::cout << one << newline; // prints 1 followed by a newline
```

Section 21.2: char16_t

Version ≥ C++11

An unsigned integer type with the same size and alignment as `uint_least16_t`, which is therefore large enough to hold a UTF-16 code unit.

```
const char16_t message[] = u"你好，世界\\n";            // Chinese for "hello, world\\n"
std::cout << sizeof(message)/sizeof(char16_t) << "\\n"; // prints 7
```

Section 21.3: char32_t

Version ≥ C++11

An unsigned integer type with the same size and alignment as `uint_least32_t`, which is therefore large enough to hold a UTF-32 code unit.

```
const char32_t full_house[] = U"        ";               // non-BMP characters
std::cout << sizeof(full_house)/sizeof(char32_t) << "\\n"; // prints 6
```

Section 21.4: int

Denotes a signed integer type with "the natural size suggested by the architecture of the execution environment", whose range includes at least -32767 to +32767, inclusive.

```
int x = 2;
int y = 3;
int z = x + y;
```

Can be combined with `unsigned`, `short`, `long`, and `long long` (q.v.) in order to yield other integer types.

Section 21.5: void

An incomplete type; it is not possible for an object to have type `void`, nor are there arrays of `void` or references to `void`. It is used as the return type of functions that do not return anything.

Moreover, a function may redundantly be declared with a single parameter of type `void`; this is equivalent to declaring a function with no parameters (e.g. `int main()` and `int main(void)` declare the same function). This syntax is allowed for compatibility with C (where function declarations have a different meaning than in C++).

The type void* ("pointer to void") has the property that any object pointer can be converted to it and back and result in the same pointer. This feature makes the type void* suitable for certain kinds of (type-unsafe) type-erasing interfaces, for example for generic contexts in C-style APIs (e.g. qsort, pthread_create).

Any expression may be converted to an expression of type void; this is called a *discarded-value expression*:

```
static_cast<void>(std::printf("Hello, %s!\n", name));  // discard return value
```

This may be useful to signal explicitly that the value of an expression is not of interest and that the expression is to be evaluated for its side effects only.

Section 21.6: wchar_t

An integer type large enough to represent all characters of the largest supported extended character set, also known as the wide-character set. (It is not portable to make the assumption that wchar_t uses any particular encoding, such as UTF-16.)

It is normally used when you need to store characters over ASCII 255 , as it has a greater size than the character type char.

```
const wchar_t message_ahmaric[] = L"           \\n"; //Ahmaric for "hello, world\\n"
const wchar_t message_chinese[] = L"你好，世界\\n";// Chinese for "hello, world\\n"
const wchar_t message_hebrew[]  = L"שלום עולם\\n"; //Hebrew for "hello, world\\n"
const wchar_t message_russian[] = L"Привет мир\\n";  //Russian for "hello, world\\n"
const wchar_t message_tamil[]   = L"ஹலோ உலகம்\\n"; //Tamil for "hello, world\\n"
```

Section 21.7: float

A floating point type. Has the narrowest range out of the three floating point types in C++.

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

Section 21.8: double

A floating point type. Its range includes that of float. When combined with long, denotes the long double floating point type, whose range includes that of double.

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

Section 21.9: long

Denotes a signed integer type that is at least as long as int, and whose range includes at least -2147483647 to +2147483647, inclusive (that is, -(2^31 - 1) to +(2^31 - 1)). This type can also be written as long int.

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

The combination long double denotes a floating point type, which has the widest range out of the three floating point types.

```
long double area(long double radius) {
    const long double pi = 3.1415926535897932385L;
    return pi*radius*radius;
}
```

Version ≥ C++11

When the `long` specifier occurs twice, as in `long long`, it denotes a signed integer type that is at least as long as `long`, and whose range includes at least -9223372036854775807 to +9223372036854775807, inclusive (that is, -(2^63 - 1) to +(2^63 - 1)).

```
// support files up to 2 TiB
const long long max_file_size = 2LL << 40;
```

Section 21.10: short

Denotes a signed integer type that is at least as long as `char`, and whose range includes at least -32767 to +32767, inclusive. This type can also be written as `short int`.

```
// (during the last year)
short hours_worked(short days_worked) {
    return 8*days_worked;
}
```

Section 21.11: bool

An integer type whose value can be either `true` or `false`.

```
bool is_even(int x) {
    return x%2 == 0;
}
const bool b = is_even(47); // false
```

Chapter 22: Variable Declaration Keywords

Section 22.1: decltype

Version ≥ C++11

Yields the type of its operand, which is not evaluated.

- If the operand `e` is a name without any additional parentheses, then `decltype(e)` is the *declared type* of `e`.

  ```
  int x = 42;
  std::vector<decltype(x)> v(100, x); // v is a vector<int>
  ```

- If the operand `e` is a class member access without any additional parentheses, then `decltype(e)` is the *declared type* of the member accessed.

  ```
  struct S {
      int x = 42;
  };
  const S s;
  decltype(s.x) y; // y has type int, even though s.x is const
  ```

- In all other cases, `decltype(e)` yields both the type and the value category of the expression `e`, as follows:

  - If `e` is an lvalue of type `T`, then `decltype(e)` is `T&`.
  - If `e` is an xvalue of type `T`, then `decltype(e)` is `T&&`.
  - If `e` is a prvalue of type `T`, then `decltype(e)` is `T`.

  This includes the case with extraneous parentheses.

  ```
  int f() { return 42; }
  int& g() { static int x = 42; return x; }
  int x = 42;
  decltype(f()) a = f(); // a has type int
  decltype(g()) b = g(); // b has type int&
  decltype((x)) c = x;   // c has type int&, since x is an lvalue
  ```

Version ≥ C++14

The special form `decltype(auto)` deduces the type of a variable from its initializer or the return type of a function from the `return` statements in its definition, using the type deduction rules of `decltype` rather than those of `auto`.

```
const int x = 123;
auto y = x;           // y has type int
decltype(auto) z = x; // z has type const int, the declared type of x
```

Section 22.2: const

A type specifier; when applied to a type, produces the const-qualified version of the type. See const keyword for details on the meaning of `const`.

```
const int x = 123;
x = 456;    // error
```

```cpp
int& r = x; // error

struct S {
    void f();
    void g() const;
};
const S s;
s.f(); // error
s.g(); // OK
```

Section 22.3: volatile

A type qualifier; when applied to a type, produces the volatile-qualified version of the type. Volatile qualification plays the same role as `const` qualification in the type system, but `volatile` does not prevent objects from being modified; instead, it forces the compiler to treat all accesses to such objects as side effects.

In the example below, if `memory_mapped_port` were not volatile, the compiler could optimize the function so that it performs only the final write, which would be incorrect if `sizeof(int)` is greater than 1. The `volatile` qualification forces it to treat all `sizeof(int)` writes as different side effects and hence perform all of them (in order).

```cpp
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

Section 22.4: signed

A keyword that is part of certain integer type names.

- When used alone, `int` is implied, so that `signed`, `signed int`, and `int` are the same type.
- When combined with `char`, yields the type `signed char`, which is a different type from `char`, even if `char` is also signed. `signed char` has a range that includes at least -127 to +127, inclusive.
- When combined with `short`, `long`, or `long long`, it is redundant, since those types are already signed.
- `signed` cannot be combined with `bool`, `wchar_t`, `char16_t`, or `char32_t`.

Example:

```cpp
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
    std::cout << "cold day, eh?\n";
}
```

Section 22.5: unsigned

A type specifier that requests the unsigned version of an integer type.

- When used alone, `int` is implied, so `unsigned` is the same type as `unsigned int`.
- The type `unsigned char` is different from the type `char`, even if `char` is unsigned. It can hold integers up to at least 255.
- `unsigned` can also be combined with `short`, `long`, or `long long`. It cannot be combined with `bool`, `wchar_t`, `char16_t`, or `char32_t`.

Example:

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
    // note: returning invert_case_table[c] directly does the
    // wrong thing on implementations where char is a signed type
}
```

Chapter 23: Keywords

Keywords have fixed meaning defined by the C++ standard and cannot be used as identifiers. It is illegal to redefine keywords using the preprocessor in any translation unit that includes a standard library header. However, keywords lose their special meaning inside attributes.

Section 23.1: asm

The `asm` keyword takes a single operand, which must be a string literal. It has an implementation-defined meaning, but is typically passed to the implementation's assembler, with the assembler's output being incorporated into the translation unit.

The `asm` statement is a *definition*, not an *expression*, so it may appear either at block scope or namespace scope (including global scope). However, since inline assembly cannot be constrained by the rules of the C++ language, `asm` may not appear inside a `constexpr` function.

Example:

```
[[noreturn]] void halt_system() {
    asm("hlt");
}
```

Section 23.2: Different keywords

> **void C++**

1. When used as a function return type, the void keyword specifies that the function does not return a value. When used for a function's parameter list, void specifies that the function takes no parameters. When used in the declaration of a pointer, void specifies that the pointer is "universal."
2. If a pointer's type is void *, the pointer can point to any variable that is not declared with the const or volatile keyword. A void pointer cannot be dereferenced unless it is cast to another type. A void pointer can be converted into any other type of data pointer.
3. A void pointer can point to a function, but not to a class member in C++.

```
void vobject;   // C2182
void *pv;   // okay
int *pint; int i;
int main() {
pv = &i;
   // Cast optional in C required in C++
pint = (int *)pv;
```

> **Volatile C++**

1. A type qualifier that you can use to declare that an object can be modified in the program by the hardware.

```
volatile declarator ;
```

virtual C++

1. The virtual keyword declares a virtual function or a virtual base class.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

Parameters

1. **type-specifiers** Specifies the return type of the virtual member function.
2. **member-function-declarator** Declares a member function.
3. **access-specifier** Defines the level of access to the base class, public, protected or private. Can appear before or after the virtual keyword.
4. **base-class-name** Identifies a previously declared class type

this pointer

1. The this pointer is a pointer accessible only within the nonstatic member functions of a class, struct, or union type. It points to the object for which the member function is called. Static member functions do not have a this pointer.

```
this->member-identifier
```

An object's this pointer is not part of the object itself; it is not reflected in the result of a sizeof statement on the object. Instead, when a nonstatic member function is called for an object, the address of the object is passed by the compiler as a hidden argument to the function. For example, the following function call:

```
myDate.setMonth( 3 );
```

can be interpreted this way:

```
setMonth( &myDate, 3 );
```

The object's address is available from within the member function as the this pointer. Most uses of this are implicit. It is legal, though unnecessary, to explicitly use this when referring to members of the class. For example:

```
void Date::setMonth( int mn )
{
   month = mn;            // These three statements
   this->month = mn;      // are equivalent
   (*this).month = mn;
}
```

The expression *this is commonly used to return the current object from a member function: return *this; The this pointer is also used to guard against self-reference:

```
if (&Object != this) {
// do not execute in cases of self-reference
```

try, throw, and catch Statements (C++)

1. To implement exception handling in C++, you use try, throw, and catch expressions.
2. First, use a try block to enclose one or more statements that might throw an exception.
3. A throw expression signals that an exceptional condition—often, an error—has occurred in a try block. You can use an object of any type as the operand of a throw expression. Typically, this object is used to communicate information about the error. In most cases, we recommend that you use the std::exception class or one of the derived classes that are defined in the standard library. If one of those is not appropriate, we recommend that you derive your own exception class from std::exception.
4. To handle exceptions that may be thrown, implement one or more catch blocks immediately following a try block. Each catch block specifies the type of exception it can handle.

```
    MyData md;
try {
   // Code that could throw an exception
   md = GetNetworkResource();
}
catch (const networkIOException& e) {
   // Code that executes when an exception of type
   // networkIOException is thrown in the try block
   // ...
   // Log error message in the exception object
   cerr << e.what();
}
catch (const myDataFormatException& e) {
   // Code that handles another exception type
   // ...
   cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
   // ...
   if (IOSuccess == false)
      throw networkIOException("Unable to connect");
   // ...
   if (readError)
      throw myDataFormatException("Format error");
   // ...
}
```

> The code after the try clause is the guarded section of code. The throw expression throws—that is, raises—an exception. The code block after the catch clause is the exception handler. This is the handler that catches the exception that's thrown if the types in the throw and catch expressions are compatible.

```
    try {
   throw CSomeOtherException();
}
catch(...) {
   // Catch all exceptions – dangerous!!!
   // Respond (perhaps only partially) to the exception, then
   // re-throw to pass the exception to some other handler
   // ...
   throw;
}
```

> **friend (C++)**

1. In some circumstances, it is more convenient to grant member-level access to functions that are not members of a class or to all members in a separate class. Only the class implementer can declare who its friends are. A function or class cannot declare itself as a friend of any class. In a class definition, use the friend keyword and the name of a non-member function or other class to grant it access to the private and protected members of your class. In a template definition, a type parameter can be declared as a friend.
2. If you declare a friend function that was not previously declared, that function is exported to the enclosing nonclass scope.

```
class friend F
friend F;
class ForwardDeclared;// Class name is known.
class HasFriends
{
   friend int ForwardDeclared::IsAFriend();// C2039 error expected
};
```

friend functions

1. A friend function is a function that is not a member of a class but has access to the class's private and protected members.Friend functions are not considered class members; they are normal external functions that are given special access privileges.
2. Friends are not in the class's scope, and they are not called using the member-selection operators (. and –>) unless they are members of another class.
3. A friend function is declared by the class that is granting access. The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

```
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
    // Output: 0
           1
}
```

Class members as friends

```
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

Section 23.3: typename

1. When followed by a qualified name, `typename` specifies that it is the name of a type. This is often required in templates, in particular, when the nested name specifier is a dependent type other than the current instantiation. In this example, `std::decay<T>` depends on the template parameter `T`, so in order to name the nested type `type`, we need to prefix the entire qualified name with `typename`. For more deatils, see Where and why do I have to put the "template" and "typename" keywords?

   ```
   template <class T>
   auto decay_copy(T&& r) -> typename std::decay<T>::type;
   ```

2. Introduces a type parameter in the declaration of a template. In this context, it is interchangeable with `class`.

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

Version ≥ C++17

3. `typename` can also be used when declaring a template template parameter, preceding the name of the parameter, just like `class`.

   ```
   template <template <class T> typename U>
   void f() {
       U<int>::do_it();
       U<double>::do_it();
   }
   ```

Section 23.4: explicit

1. When applied to a single-argument constructor, prevents that constructor from being used to perform implicit conversions.

```
class MyVector {
  public:
    explicit MyVector(uint64_t size);
};
MyVector v1(100);  // ok
uint64_t len1 = 100;
MyVector v2{len1}; // ok, len1 is uint64_t
int len2 = 100;
MyVector v3{len2}; // ill-formed, implicit conversion from int to uint64_t
```

Since C++11 introduced initializer lists, in C++11 and later, `explicit` can be applied to a constructor with any number of arguments, with the same meaning as in the single-argument case.

```
struct S {
    explicit S(int x, int y);
};
S f() {
    return {12, 34};  // ill-formed
    return S{12, 34}; // ok
}
```

Version ≥ C++11

2. When applied to a conversion function, prevents that conversion function from being used to perform implicit conversions.

```
class C {
    const int x;
  public:
    C(int x) : x(x) {}
    explicit operator int() { return x; }
};
C c(42);
int x = c;                    // ill-formed
int y = static_cast<int>(c); // ok; explicit conversion
```

Section 23.5: sizeof

A unary operator that yields the size in bytes of its operand, which may be either an expression or a type. If the operand is an expression, it is not evaluated. The size is a constant expression of type `std::size_t`.

If the operand is a type, it must be parenthesized.

- It is illegal to apply `sizeof` to a function type.
- It is illegal to apply `sizeof` to an incomplete type, including `void`.
- If sizeof is applied to a reference type `T&` or `T&&`, it is equivalent to `sizeof(T)`.
- When `sizeof` is applied to a class type, it yields the number of bytes in a complete object of that type, including any padding bytes in the middle or at the end. Therefore, a `sizeof` expression can never have a value of 0. See layout of object types for more details.

- The char, signed char, and unsigned char types have a size of 1. Conversely, a byte is defined to be the amount of memory required to store a char object. It does not necessarily mean 8 bits, as some systems have char objects longer than 8 bits.

If *expr* is an expression, sizeof(*expr*) is equivalent to sizeof(T) where T is the type of *expr.*

```
int a[100];
std::cout << "The number of bytes in `a` is: " << sizeof a;
memset(a, 0, sizeof a); // zeroes out the array
```

Version ≥ C++11

The sizeof... operator yields the number of elements in a parameter pack.

```
template <class... T>
void f(T&&...) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

Section 23.6: noexcept

Version ≥ C++11

1. A unary operator that determines whether the evaluation of its operand can propagate an exception. Note that the bodies of called functions are not examined, so noexcept can yield false negatives. The operand is not evaluated.

```
#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << '\n'; // prints 0
    std::cout << noexcept(bar()) << '\n'; // prints 0
    std::cout << noexcept(1 + 1) << '\n'; // prints 1
    std::cout << noexcept(S()) << '\n';   // prints 1
}
```

In this example, even though bar() can never throw an exception, noexcept(bar()) is still false because the fact that bar() cannot propagate an exception has not been explicitly specified.

2. When declaring a function, specifies whether or not the function can propagate an exception. Alone, it declares that the function cannot propagate an exception. With a parenthesized argument, it declares that the function can or cannot propagate an exception depending on the truth value of the argument.

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

In this example, we have declared that `f4`, `f5`, and `f6` cannot propagate exceptions. (Although an exception can be thrown during execution of `f6`, it is caught and not allowed to propagate out of the function.) We have declared that `f2` may propagate an exception. When the `noexcept` specifier is omitted, it is equivalent to `noexcept(false)`, so we have implicitly declared that `f1` and `f3` may propagate exceptions, even though exceptions cannot actually be thrown during the execution of `f3`.

Version ≥ C++17

Whether or not a function is `noexcept` is part of the function's type: that is, in the example above, `f1`, `f2`, and `f3` have different types from `f4`, `f5`, and `f6`. Therefore, `noexcept` is also significant in function pointers, template arguments, and so on.

```
void g1() {}
void g2() noexcept {}
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept
void (*p2)() noexcept = &g2; // ok; types match
void (*p3)() = &g1;          // ok; types match
void (*p4)() = &g2;          // ok; implicit conversion
```

Chapter 24: Returning several values from a function

There are many situations where it is useful to return several values from a function: for example, if you want to input an item and return the price and number in stock, this functionality could be useful. There are many ways to do this in C++, and most involve the STL. However, if you wish to avoid the STL for some reason, there are still several ways to do this, including `structs/classes` and `arrays`.

Section 24.1: Using std::tuple

Version ≥ C++11

The type `std::tuple` can bundle any number of values, potentially including values of different types, into a single return object:

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
   return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

In C++17, a braced initializer list can be used:

Version ≥ C++17

```
std::tuple<int, int, int, int> foo(int a, int b)    {
    return {a + b, a - b, a * b, a / b};
}
```

Retrieving values from the returned `tuple` can be cumbersome, requiring the use of the `std::get` template function:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

If the types can be declared before the function returns, then `std::tie` can be employed to unpack a `tuple` into existing variables:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

If one of the returned values is not needed, `std::ignore` can be used:

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

Version ≥ C++17

Structured bindings can be used to avoid `std::tie`:

```
auto [add, sub, mul, div] = foo(5,12);
```

If you want to return a tuple of lvalue references instead of a tuple of values, use `std::tie` in place of `std::make_tuple`.

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
  if (b<a)
    return std::tie(b,a);
  else
    return std::tie(a,b);
}
```

which permits

```
void increase_least(int& a, int& b) {
  std::get<0>(minmax(a,b))++;
}
```

In some rare cases you'll use `std::forward_as_tuple` instead of `std::tie`; be careful if you do so, as temporaries may not last long enough to be consumed.

Section 24.2: Structured Bindings

Version ≥ C++17

C++17 introduces structured bindings, which makes it even easier to deal with multiple return types, as you do not need to rely upon `std::tie()` or do any manual tuple unpacking:

```
std::map<std::string, int> m;

// insert an element into the map and check if insertion succeeded
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // your code goes here
}

// iterate over all elements without having to use the cryptic 'first' and 'second' names
for (auto const& [key, value] : m) {
    std::cout << "The value for " << key << " is " << value << '\n';
}
```

Structured bindings can be used by default with `std::pair`, `std::tuple`, and any type whose non-static data members are all either public direct members or members of an unambiguous base class:

```
struct A { int x; };
struct B : A { int y; };
B foo();

// with structured bindings
const auto [x, y] = foo();

// equivalent code without structured bindings
const auto result = foo();
auto& x = result.x;
auto& y = result.y;
```

If you make your type "tuple-like" it will also automatically work with your type. A tuple-like is a type with appropriate `tuple_size`, `tuple_element` and `get` written:

```
namespace my_ns {
    struct my_type {
        int x;
        double d;
        std::string s;
    };
    struct my_type_view {
        my_type* ptr;
    };
}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}
```

now this works:

```
my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << '\n';
}
```

Section 24.3: Using struct

A struct can be used to bundle multiple return values:

Version ≥ C++11

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}
```

```
auto calc = foo(5, 12);
```

Version < C++11

Instead of assignment to individual fields, a constructor can be used to simplify the constructing of returned values:

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
    foo_return_type(int add, int sub, int mul, int div)
    : add(add), sub(sub), mul(mul), div(div) {}
};

foo_return_type foo(int a, int b) {
     return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

The individual results returned by the function `foo()` can be retrieved by accessing the member variables of the `struct` `calc`:

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n';
```

Output:

```
17 -7 60 0
```

Note: When using a `struct`, the returned values are grouped together in a single object and accessible using meaningful names. This also helps to reduce the number of extraneous variables created in the scope of the returned values.

Version ≥ C++17

In order to unpack a `struct` returned from a function, structured bindings can be used. This places the out-parameters on an even footing with the in-parameters:

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n';
```

The output of this code is identical to that above. The `struct` is still used to return the values from the function. This permits you do deal with the fields individually.

Section 24.4: Using Output Parameters

Parameters can be used for returning one or more values; those parameters are required to be non-`const` pointers or references.

References:

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {
    c = a + b;
    d = a - b;
    e = a * b;
    f = a / b;
}
```

Pointers:

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {
    *c = a + b;
    *d = a - b;
    *e = a * b;
    *f = a / b;
}
```

Some libraries or frameworks use an empty 'OUT' `#define` to make it abundantly obvious which parameters are output parameters in the function signature. This has no functional impact, and will be compiled out, but makes the function signature a bit clearer;

```
#define OUT

void calculate(int a, int b, OUT int& c) {
    c = a + b;
}
```

Section 24.5: Using a Function Object Consumer

We can provide a consumer that will be called with the multiple relevant values:

Version ≥ C++11

```
template <class F>
void foo(int a, int b, F consumer) {
    consumer(a + b, a - b, a * b, a / b);
}

// use is simple... ignoring some results is possible as well
foo(5, 12, [](int sum, int , int , int ){
    std::cout << "sum is " << sum << '\n';
});
```

This is known as "continuation passing style".

You can adapt a function returning a tuple into a continuation passing style function via:

Version ≥ C++17

```
template<class Tuple>
struct continuation {
  Tuple t;
  template<class F>
  decltype(auto) operator->*(F&& f)&&{
    return std::apply( std::forward<F>(f), std::move(t) );
  }
};
std::tuple<int,int,int,int> foo(int a, int b);

continuation(foo(5,12))->*[](int sum, auto&&...) {
  std::cout << "sum is " << sum << '\n';
};
```

with more complex versions being writable in C++14 or C++11.

Section 24.6: Using std::pair

The struct template `std::pair` can bundle together *exactly* two return values, of any two types:

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

With C++11 or later, an initializer list can be used instead of `std::make_pair`:

Version ≥ C++11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

The individual values of the returned `std::pair` can be retrieved by using the pair's `first` and `second` member objects:

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

Output:

> 10

Section 24.7: Using std::array

Version ≥ C++11

The container `std::array` can bundle together a fixed number of return values. This number has to be known at compile-time and all return values have to be of the same type:

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

This replaces c style arrays of the form `int bar[4]`. The advantage being that various `c++` std functions can now be used on it. It also provides useful member functions like `at` which is a safe member access function with bound checking, and `size` which allows you to return the size of the array without calculation.

Section 24.8: Using Output Iterator

Several values of the same type can be returned by passing an output iterator to the function. This is particularly common for generic functions (like the algorithms of the standard library).

Example:

```
template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
}
```

Example usage:

```
std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits now contains {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Section 24.9: Using std::vector

A `std::vector` can be useful for returning a dynamic number of variables of the same type. The following example uses `int` as data type, but a `std::vector` can hold any type that is trivially copyable:

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

// the following function returns all integers between and including 'a' and 'b' in a vector
// (the function can return up to std::vector::max_size elements with the vector, given that
// the system's main memory can hold that many items)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // assigns the filled vector created inside the function to the new vector 'v'
    std::vector<int> v = fillVectorFrom(1, 10);

    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Chapter 25: Polymorphism

Section 25.1: Define polymorphic classes

The typical example is an abstract shape class, that can then be derived into squares, circles, and other concrete shapes.

The parent class:

Let's start with the polymorphic class:

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual double get_surface() const = 0;
    virtual void describe_object() const { std::cout << "this is a shape" << std::endl; }

    double get_doubled_surface() const { return 2 * get_surface(); }
};
```

How to read this definition ?

- You can define polymorphic behavior by introduced member functions with the keyword `virtual`. Here `get_surface()` and `describe_object()` will obviously be implemented differently for a square than for a circle. When the function is invoked on an object, function corresponding to the real class of the object will be determined at runtime.
- It makes no sense to define `get_surface()` for an abstract shape. This is why the function is followed by `= 0`. This means that the function is *pure virtual function*.
- A polymorphic class should always define a virtual destructor.
- You may define non virtual member functions. When these function will be invoked for an object, the function will be chosen depending on the class used at compile-time. Here `get_double_surface()` is defined in this way.
- A class that contains at least one pure virtual function is an abstract class. Abstract classes cannot be instantiated. You may only have pointers or references of an abstract class type.

Derived classes

Once a polymorphic base class is defined you can derive it. For example:

```
class Square : public Shape {
    Point top_left;
    double side_length;
public:
    Square (const Point& top_left, double side)
       : top_left(top_left), side_length(side_length) {}

    double get_surface() override { return side_length * side_length; }
    void describe_object() override {
        std::cout << "this is a square starting at " << top_left.x << ", " << top_left.y
                  << " with a length of " << side_length << std::endl;
    }
};
```

Some explanations:

- You can define or override any of the virtual functions of the parent class. The fact that a function was virtual in the parent class makes it virtual in the derived class. No need to tell the compiler the keyword `virtual` again. But it's recommended to add the keyword `override` at the end of the function declaration, in order to prevent subtle bugs caused by unnoticed variations in the function signature.
- If all the pure virtual functions of the parent class are defined you can instantiate objects for this class, else it will also become an abstract class.
- You are not obliged to override all the virtual functions. You can keep the version of the parent if it suits your need.

Example of instantiation

```
int main() {

    Square square(Point(10.0, 0.0), 6); // we know it's a square, the compiler also
    square.describe_object();
    std::cout << "Surface: " << square.get_surface() << std::endl;

    Circle circle(Point(0.0, 0.0), 5);

    Shape *ps = nullptr;  // we don't know yet the real type of the object
    ps = &circle;         // it's a circle, but it could as well be a square
    ps->describe_object();
    std::cout << "Surface: " << ps->get_surface() << std::endl;
}
```

Section 25.2: Safe downcasting

Suppose that you have a pointer to an object of a polymorphic class:

```
Shape *ps;                       // see example on defining a polymorphic class
ps =  get_a_new_random_shape();  // if you don't have such a function yet, you
                                 // could just write ps = new Square(0.0,0.0, 5);
```

a downcast would be to cast from a general polymorphic `Shape` down to one of its derived and more specific shape like `Square` or `Circle`.

Why to downcast ?

Most of the time, you would not need to know which is the real type of the object, as the virtual functions allow you to manipulate your object independently of its type:

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

If you don't need any downcast, your design would be perfect.

However, you may need sometimes to downcast. A typical example is when you want to invoke a non virtual function that exist only for the child class.

Consider for example circles. Only circles have a diameter. So the class would be defined as :

```
class Circle: public Shape { // for Shape, see example on defining a polymorphic class
    Point center;
    double radius;
public:
    Circle (const Point& center, double radius)
       : center(center), radius(radius) {}

    double get_surface() const override { return r * r * M_PI; }

    // this is only for circles. Makes no sense for other shapes
    double get_diameter() const { return 2 * r; }
};
```

The `get_diameter()` member function only exist for circles. It was not defined for a `Shape` object:

```
Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error
```

How to downcast ?

If you'd know for sure that `ps` points to a circle you could opt for a `static_cast`:

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

This will do the trick. But it's very risky: if `ps` appears to by anything else than a `Circle` the behavior of your code will be undefined.

So rather than playing Russian roulette, you should safely use a `dynamic_cast`. This is specifically for polymorphic classes :

```
int main() {
    Circle circle(Point(0.0, 0.0), 10);
    Shape &shape = circle;

    std::cout << "The shape has a surface of " << shape.get_surface() << std::endl;

    //shape.get_diameter();   // OUCH !!! Compilation error

    Circle *pc = dynamic_cast<Circle*>(&shape); // will be nullptr if ps wasn't a circle
    if (pc)
        std::cout << "The shape is a circle of diameter " << pc->get_diameter() << std::endl;
    else
        std::cout << "The shape isn't a circle !" << std::endl;
}
```

Note that `dynamic_cast` is not possible on a class that is not polymorphic. You'd need at least one virtual function in the class or its parents to be able to use it.

Section 25.3: Polymorphism & Destructors

If a class is intended to be used polymorphically, with derived instances being stored as base pointers/references, its base class' destructor should be either `virtual` or `protected`. In the former case, this will cause object destruction to check the `vtable`, automatically calling the correct destructor based on the dynamic type. In the latter case, destroying the object through a base class pointer/reference is disabled, and the object can only be deleted when explicitly treated as its actual type.

```
struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};
struct ProtectedDestructor {
  protected:
    ~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // Looks up VirtualDestructor::~VirtualDestructor() in vtable, sees it's
           // VirtualDerived::~VirtualDerived(), calls that.

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // Error: ProtectedDestructor::~ProtectedDestructor() is protected.
delete static_cast<ProtectedDerived*>(pd); // Good.
```

Both of these practices guarantee that the derived class' destructor will always be called on derived class instances, preventing memory leaks.

Chapter 26: References

Section 26.1: Defining a reference

References behaves similarly, but not entirely like const pointers. A reference is defined by suffixing an ampersand `&` to a type name.

```
int i = 10;
int &refi = i;
```

Here, `refi` is a reference bound to `i`.
References abstracts the semantics of pointers, acting like an alias to the underlying object:

```
refi = 20; // i = 20;
```

You can also define multiple references in a single definition:

```
int i = 10, j = 20;
int &refi = i, &refj = j;

// Common pitfall :
// int& refi = i, k = j;
// refi will be of type int&.
// though, k will be of type int, not int&!
```

References **must** be initialized correctly at the time of definition, and cannot be modified afterwards. The following piece of codes causes a compile error:

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

You also cannot bind directly a reference to `nullptr`, unlike pointers:

```
int *const ptri = nullptr;
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a temporary
of type 'nullptr_t'
```

Chapter 27: Value and Reference Semantics

Section 27.1: Definitions

A type has value semantics if the object's observable state is functionally distinct from all other objects of that type. This means that if you copy an object, you have a new object, and modifications of the new object will not be in any way visible from the old object.

Most basic C++ types have value semantics:

```
int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.
```

Most standard-library defined types have value semantics too:

```
std::vector<int> v1(5, 12); //array of 5 values, 12 in each.
std::vector<int> v2 = v1; //Copies the vector.
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //Writes "12 12", since v1 is unchanged.
```

A type is said to have reference semantics if an instance of that type can share its observable state with another object (external to it), such that manipulating one object will cause the state to change within another object.

C++ pointers have value semantics with regard to which object they point to, but they have reference semantics with regard to the *state* of the object they point to:

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //Will always pass.

int *pj = pi;
*pj += 5;
std::cout << *pi; //Writes 9, since `pi` and `pj` reference the same object.
```

C++ references have reference semantics as well.

Section 27.2: Deep copying and move support

If a type wishes to have value semantics, and it needs to store objects that are dynamically allocated, then on copy operations, the type will need to allocate new copies of those objects. It must also do this for copy assignment.

This kind of copying is called a "deep copy". It effectively takes what would have otherwise been reference semantics and turns it into value semantics:

```
struct Inner {int i;};

const int NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.
public:
  Value() : array_(new Inner[NUM_INNER]){}

  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }
};
```

Version ≥ C++11

Move semantics allow a type like `Value` to avoid truly copying its referenced data. If the user uses the value in a way that provokes a move, the "copied" from object can be left empty of the data it referenced:

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

Indeed, we can even make such a type non-copyable, if we want to forbid deep copies while still allowing the object to be moved around.

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) = delete;
  Value &operator=(const Value &val) = delete;

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

We can even apply the Rule of Zero, through the use of `unique_ptr`:

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  unique_ptr<Inner []>array_; //Move-only type.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //No need to explicitly delete. Or even declare.
  ~Value() = default; {delete[] array_;}

  //No need to explicitly delete. Or even declare.
  Value(const Value &val) = default;
  Value &operator=(const Value &val) = default;

  //Will perform an element-wise move.
  Value(Value &&val) noexcept = default;

  //Will perform an element-wise move.
  Value &operator=(Value &&val) noexcept = default;
};
```

Chapter 28: C++ function "call by value" vs. "call by reference"

The scope of this section is to explain the differences in theory and implementation for what happens with the parameters of a function upon calling.

In detail the parameters can be seen as variables before the function call and inside the function, where the visible behaviour and accessibility to these variables differs with the method used to hand them over.

Additionally, the reusability of variables and their respective values after the function call also is explained by this topic.

Section 28.1: Call by value

Upon calling a function there are new elements created on the program stack. These include some information about the function and also space (memory locations) for the parameters and the return value.

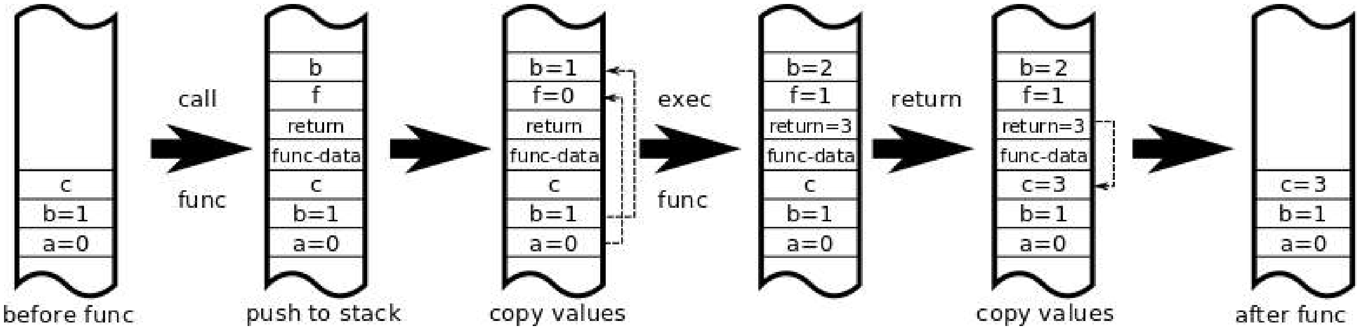
When handing over a parameter to a function the value of the used variable (or literal) is copied into the memory location of the function parameter. This implies that now there a two memory locations with the same value. Inside of the function we only work on the parameter memory location.

After leaving the function the memory on the program stack is popped (removed) which erases all data of the function call, including the memory location of the parameters we used inside. Thus, the values changed inside the function do not affect the outside variables values.

```
int func(int f, int b) {
  //new variables are created and values from the outside copied
  //f has a value of 0
  //inner_b has a value of 1
  f = 1;
  //f has a value of 1
  b = 2;
  //inner_b has a value of 2
  return f+b;
}

int main(void) {
  int a = 0;
  int b = 1; //outer_b
  int c;

  c = func(a,b);
  //the return value is copied to c

  //a has a value of 0
  //outer_b has a value of 1   <--- outer_b and inner_b are different variables
  //c has a value of 3
}
```

In this code we create variables inside the main function. These get assigned values. Upon calling the functions there are two new variables created: `f` and `inner_b` where `b` shares the name with the outer variable it does not share the memory location. The behaviour of `a<->f` and `b<->b` is identical.

The following graphic symbolizes what is happening on the stack and why there is no change in varibale `b`. The graphic is not fully accurate but emphazises the example.



It is called "call by value" because we do not hand over the variables but only the values of these variables.

Chapter 29: Copying vs Assignment

rhs	Right Hand Side of the equality for both copy and assignment constructors. For example the assignment constructor : MyClass operator=(MyClass& rhs);
Placeholder	Placeholder

Section 29.1: Assignment Operator

The Assignment Operator is when you replace the data with an already existing(previously initialized) object with some other object's data. Lets take this as an example:

```
// Assignment Operator
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo& operator=(const Foo& rhs)
    {
            data = rhs.data;
            return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2(42);
   foo = foo2; // Assignment Operator Called
   cout << foo.data << endl; //Prints 42
}
```

You can see here I call the assignment operator when I already initialized the `foo` object. Then later I assign `foo2` to `foo` . All the changes to appear when you call that equal sign operator is defined in your `operator=` function. You can see a runnable output here: http://cpp.sh/3qtbm

Section 29.2: Copy Constructor

Copy constructor on the other hand , is the complete opposite of the Assignment Constructor. This time, it is used to initialize an already nonexistent(or non-previously initialized) object. This means it copies all the data from the object you are assigning it to , without actually initializing the object that is being copied onto. Now Let's take a look at the same code as before but modify the assignment constructor to be a copy constructor :

```
// Copy Constructor
#include <iostream>
#include <string>
```

```
using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2 = foo; // Copy Constructor called
   cout << foo2.data << endl;
}
```

You can see here `Foo foo2 = foo;` in the main function I immediately assign the object before actually initializing it, which as said before means it's a copy constructor. And notice that I didn't need to pass the parameter int for the `foo2` object since I automatically pulled the previous data from the object foo. Here is an example output : http://cpp.sh/5iu7

Section 29.3: Copy Constructor Vs Assignment Constructor

Ok we have briefly looked over what the copy constructor and assignment constructor are above and gave examples of each now let's see both of them in the same code. This code will be similar as above two. Let's take this :

```
// Copy vs Assignment Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
        return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) / Normal Constructor called
   Foo foo2 = foo; //Copy Constructor Called
   cout << foo2.data << endl;

   Foo foo3(42);
   foo3=foo; //Assignment Constructor Called
   cout << foo3.data << endl;
}
```

Output:

```
2
2
```

Here you can see we first call the copy constructor by executing the line `Foo foo2 = foo;` . Since we didn't initialize it previously. And then next we call the assignment operator on foo3 since it was already initialized `foo3=foo`;

Chapter 30: Pointers

A pointer is an address that refers to a location in memory. They're commonly used to allow functions or data structures to know of and modify memory without having to copy the memory referred to. Pointers are usable with both primitive (built-in) or user-defined types.

Pointers make use of the "dereference" `*` , "address of" `&` , and "arrow" `->` operators. The '*' and '->' operators are used to access the memory being pointed at, and the `&` operator is used to get an address in memory.

Section 30.1: Pointer Operations

There are two operators for pointers: Address-of operator (&): Returns the memory address of its operand. Contents-of (Dereference) operator(*): Returns the value of the variable located at the address specified by its operator.

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//Outputs 20 (The value of var)

cout << ptr << endl;
//Outputs 0x234f119 (var's memory location)

cout << *ptr << endl;
//Outputs 20(The value of the variable stored in the pointer ptr
```

The asterisk (*) is used in declaring a pointer for simple purpose of indicating that it is a pointer. Don't confuse this with the **dereference** operator, which is used to obtain the value located at the specified address. They are simply two different things represented with the same sign.

Section 30.2: Pointer basics

Version < C++11

Note: in all the following, the existence of the C++11 constant `nullptr` is assumed. For earlier versions, replace `nullptr` with `NULL`, the constant that used to play a similar role.

Creating a pointer variable

A pointer variable can be created using the specific `*` syntax, e.g. `int *pointer_to_int;`.
When a variable is of a *pointer type* (`int *`), it just contains a memory address. The memory address is the location at which data of the *underlying type* (`int`) is stored.

The difference is clear when comparing the size of a variable with the size of a pointer to the same type:

```
// Declare a struct type `big_struct` that contains
// three long long ints.
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// Create a variable `bar` of type `big_struct`
```

```
big_struct bar;
// Create a variable `p_bar` of type `pointer to big_struct`.
// Initialize it to `nullptr` (a null pointer).
big_struct *p_bar0 = nullptr;

// Print the size of `bar`
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// Print the size of `p_bar`.
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* Produces:
    sizeof(bar) = 24
    sizeof(p_bar0) = 8
*/
```

Taking the address of another variable

Pointers can be assigned between each other just as normal variables; in this case, it is the **memory address** that is copied from one pointer to another, **not the actual data** that a pointer points to.
Moreover, they can take the value `nullptr` which represents a null memory location. A pointer equal to `nullptr` contains an invalid memory location and hence it does not refer to valid data.

You can get the memory address of a variable of a given type by prefixing the variable with the *address of* operator `&`. The value returned by `&` is a pointer to the underlying type which contains the memory address of the variable (which is valid data **as long as the variable does not go out of scope**).

```
// Copy `p_bar0` into `p_bar_1`.
big_struct *p_bar1 = p_bar0;

// Take the address of `bar` into `p_bar_2`
big_struct *p_bar2 = &bar;

// p_bar1 is now nullptr, p_bar2 is &bar.

p_bar0 = p_bar2;

// p_bar0 is now &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr
```

In contrast with references:

- assigning two pointers does not overwrite the memory that the assigned pointer refers to;
- pointers can be null.
- the *address of* operator is required explicitly.

Accessing the content of a pointer

As taking an address requires `&`, as well accessing the content requires the usage of the *dereference operator* `*`, as a prefix. When a pointer is dereferenced, it becomes a variable of the underlying type (actually, a reference to it). It can then be read and modified, if not `const`.

```
(*p_bar0).foo1 = 5;

// `p_bar0` points to `bar`. This prints 5.
```

```
std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// Assign the value pointed to by `p_bar0` to `baz`.
big_struct baz;
baz = *p_bar0;

// Now `baz` contains a copy of the data pointed to by `p_bar0`.
// Indeed, it contains a copy of `bar`.

// Prints 5 as well
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;
```

The combination of `*` and the operator `.` is abbreviated by `->`:

```
std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // Prints 5
std::cout << "bar.foo1 = " <<  p_bar0->foo1  << std::endl; // Prints 5
```

Dereferencing invalid pointers

When dereferencing a pointer, you should make sure it points to valid data. Dereferencing an invalid pointer (or a null pointer) can lead to memory access violation, or to read or write garbage data.

```
big_struct *never_do_this() {
   // This is a local variable. Outside `never_do_this` it doesn't exist.
   big_struct retval;
   retval.foo1 = 11;
   // This returns the address of `retval`.
   return &retval;
   // `retval` is destroyed and any code using the value returned
   // by `never_do_this` has a pointer to a memory location that
   // contains garbage data (or is inaccessible).
}
```

In such scenario, `g++` and `clang++` correctly issue the warnings:

```
(Clang) warning: address of stack memory associated with local variable 'retval' returned [-
Wreturn-stack-address]
(Gcc)   warning: address of local variable ‘retval’ returned [-Wreturn-local-addr]
```

Hence, care must be taken when pointers are arguments of functions, as they could be null:

```
void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->foo1 = 12;
}

// Segmentation fault.
naive_code(nullptr);
```

Section 30.3: Pointer Arithmetic

Increment / Decrement

A pointer can be incremented or decremented (prefix and postfix). Incrementing a pointer advances the pointer value to the element in the array one element past the currently pointed to element. Decrementing a pointer moves it to the previous element in the array.

Pointer arithmetic is not permitted if the type that the pointer points to is not complete. void is always an incomplete type.

```
char* str = new char[10]; // str = 0x010
++str;                    // str = 0x011  in this case sizeof(char) = 1 byte

int* arr = new int[10];   // arr = 0x00100
++arr;                    // arr = 0x00104 if sizeof(int) = 4 bytes

void* ptr = (void*)new char[10];
++ptr;    // void is incomplete.
```

If a pointer to the end element is incremented, then the pointer points to one element past the end of the array. Such a pointer cannot be dereferenced, but it can be decremented.

Incrementing a pointer to the one-past-the-end element in the array, or decrementing a pointer to the first element in an array yields undefined behavior.

A pointer to a non-array object can be treated, for the purposes of pointer arithmetic, as though it were an array of size 1.

Addition / Subtraction

Integer values can be added to pointers; they act as incrementing, but by a specific number rather than by 1. Integer values can be subtracted from pointers as well, acting as pointer decrementing. As with incrementing/decrementing, the pointer must point to a complete type.

```
char* str = new char[10];  // str = 0x010
str += 2;                  // str = 0x010 + 2 * sizeof(char) = 0x012

int* arr = new int[10];    // arr = 0x100
arr += 2;                  // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) == 4.
```

Pointer Differencing

The difference between two pointers to the same type can be computed. The two pointers must be within the same array object; otherwise undefined behavior results.

Given two pointers P and Q in the same array, if P is the ith element in the array, and Q is the jth element, then P - Q shall be i - j. The type of the result is std::ptrdiff_t, from **<cstddef>**.

```
char* start = new char[10];  // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; //Equal to 5.
std::ptrdiff_t diff = start - test; //Equal to -5; ptrdiff_t is signed.
```

Chapter 31: Pointers to members

Section 31.1: Pointers to static member functions

A `static` member function is just like an ordinary C/C++ function, except with scope:

- It is inside a `class`, so it needs its name decorated with the class name;
- It has accessibility, with `public`, `protected` or `private`.

So, if you have access to the `static` member function and decorate it correctly, then you can point to the function like any normal function outside a `class`:

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

// Note that MyFn() is of type 'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
    // Note that Static() is of type 'Fn'
    static int Static(int i) { return 3*i; }
}; // Class

int main() {
    Fn *fn;    // fn is a pointer to a type-of Fn

    fn = &MyFn;          // Point to one function
    fn(3);               // Call it
    fn = &Class::Static; // Point to the other function
    fn(4);               // Call it
 } // main()
```

Section 31.2: Pointers to member functions

To access a member function of a class, you need to have a "handle" to the particular instance, as either the instance itself, or a pointer or reference to it. Given a class instance, you can point to various of its members with a pointer-to-member, IF you get the syntax correct! Of course, the pointer has to be declared to be of the same type as what you are pointing to...

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

class Class {
public:
    // Note that A() is of type 'Fn'
    int A(int a) { return 2*a; }
    // Note that B() is of type 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c;          // Need a Class instance to play with
    Class *p = &c;    // Need a Class pointer to play with

    Fn Class::*fn;    // fn is a pointer to a type-of Fn within Class

    fn = &Class::A;   // fn now points to A within any Class
    (c.*fn)(5);       // Pass 5 to c's function A (via fn)
    fn = &Class::B;    // fn now points to B within any Class
    (p->*fn)(6);       // Pass 6 to c's (via p) function B (via fn)
} // main()
```

Unlike pointers to member variables (in the previous example), the association between the class instance and the member pointer need to be bound tightly together with parentheses, which looks a little strange (as though the `.*` and `->*` aren't strange enough!)

Section 31.3: Pointers to member variables

To access a member of a `class`, you need to have a "handle" to the particular instance, as either the instance itself, or a pointer or reference to it. Given a `class` instance, you can point to various of its members with a pointer-to-member, IF you get the syntax correct! Of course, the pointer has to be declared to be of the same type as what you are pointing to...

```
class Class {
public:
    int x, y, z;
    char m, n, o;
}; // Class

int x;  // Global variable

int main() {
    Class c;        // Need a Class instance to play with
    Class *p = &c;  // Need a Class pointer to play with

    int *p_i;       // Pointer to an int

    p_i = &x;       // Now pointing to x
    p_i = &c.x;     // Now pointing to c's x

    int Class::*p_C_i; // Pointer to an int within Class

    p_C_i = &Class::x; // Point to x within any Class
    int i = c.*p_C_i;  // Use p_c_i to fetch x from c's instance
    p_C_i = &Class::y; // Point to y within any Class
    i = c.*p_C_i;      // Use p_c_i to fetch y from c's instance

    p_C_i = &Class::m; // ERROR! m is a char, not an int!

    char Class::*p_C_c = &Class::m; // That's better...
} // main()
```

The syntax of pointer-to-member requires some extra syntactic elements:

- To define the type of the pointer, you need to mention the base type, as well as the fact that it is inside a class: `int Class::*ptr;`.
- If you have a class or reference and want to use it with a pointer-to-member, you need to use the `.*` operator (akin to the `.` operator).
- If you have a pointer to a class and want to use it with a pointer-to-member, you need to use the `->*` operator (akin to the `->` operator).

Section 31.4: Pointers to static member variables

A `static` member variable is just like an ordinary C/C++ variable, except with scope:

- It is inside a `class`, so it needs its name decorated with the class name;
- It has accessibility, with `public`, `protected` or `private`.

So, if you have access to the `static` member variable and decorate it correctly, then you can point to the variable like any normal variable outside a `class`:

```
class Class {
public:
    static int i;
}; // Class

int Class::i = 1; // Define the value of i (and where it's stored!)

int j = 2;   // Just another global variable

int main() {
    int k = 3; // Local variable

    int *p;

    p = &k;   // Point to k
    *p = 2;   // Modify it
    p = &j;   // Point to j
    *p = 3;   // Modify it
    p = &Class::i; // Point to Class::i
    *p = 4;   // Modify it
} // main()
```

Chapter 32: The This Pointer

Section 32.1: this Pointer

All non-static member functions have a hidden parameter, a pointer to an instance of the class, named `this`; this parameter is silently inserted at the beginning of the parameter list, and handled entirely by the compiler. When a member of the class is accessed inside a member function, it is silently accessed through `this`; this allows the compiler to use a single non-static member function for all instances, and allows a member function to call other member functions polymorphically.

```
struct ThisPointer {
    int i;

    ThisPointer(int ii);

    virtual void func();

    int  get_i() const;
    void set_i(int ii);
};
ThisPointer::ThisPointer(int ii) : i(ii) {}
// Compiler rewrites as:
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// Constructor is responsible for turning allocated memory into 'this'.
// As the constructor is responsible for creating the object, 'this' will not be "fully"
// valid until the instance is fully constructed.

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}
// Compiler rewrites as:
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int  ThisPointer::get_i() const { return i; }
// Compiler rewrites as:
int  ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// Compiler rewrites as:
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }
```

In a constructor, `this` can safely be used to (implicitly or explicitly) access any field that has already been initialised, or any field in a parent class; conversely, (implicitly or explicitly) accessing any fields that haven't yet been initialised, or any fields in a derived class, is unsafe (due to the derived class not yet being constructed, and thus its fields neither being initialised nor existing). It is also unsafe to call virtual member functions through `this` in the constructor, as any derived class functions will not be considered (due to the derived class not yet being constructed, and thus its constructor not yet updating the vtable).

Also note that while in a constructor, the type of the object is the type which that constructor constructs. This holds true even if the object is declared as a derived type. For example, in the below example, `ctd_good` and `ctd_bad` are type `CtorThisBase` inside `CtorThisBase()`, and type `CtorThis` inside `CtorThis()`, even though their canonical type is `CtorThisDerived`. As the more-derived classes are constructed around the base class, the instance gradually goes through the class hierarchy until it is a fully-constructed instance of its intended type.

```
class CtorThisBase {
    short s;

  public:
    CtorThisBase() : s(516) {}
};

class CtorThis : public CtorThisBase {
    int i, j, k;

  public:
    // Good constructor.
    CtorThis() : i(s + 42), j(this->i), k(j) {}

    // Bad constructor.
    CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
        virt_func();
    }

    virtual void virt_func() { i += 2; }
};

class CtorThisDerived : public CtorThis {
    bool b;

  public:
    CtorThisDerived()       : b(true) {}
    CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }
};

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);
```

With these classes and member functions:

- In the good constructor, for `ctd_good`:
  - `CtorThisBase` is fully constructed by the time the `CtorThis` constructor is entered. Therefore, `s` is in a valid state while initialising `i`, and can thus be accessed.
  - `i` is initialised before `j(this->i)` is reached. Therefore, `i` is in a valid state while initialising `j`, and can thus be accessed.
  - `j` is initialised before `k(j)` is reached. Therefore, `j` is in a valid state while initialising `k`, and can thus be accessed.
- In the bad constructor, for `ctd_bad`:
  - `k` is initialised after `j(this->k)` is reached. Therefore, `k` is in an invalid state while initialising `j`, and accessing it causes undefined behaviour.
  - `CtorThisDerived` is not constructed until after `CtorThis` is constructed. Therefore, `b` is in an invalid state while initialising `k`, and accessing it causes undefined behaviour.
  - The object `ctd_bad` is still a `CtorThis` until it leaves `CtorThis()`, and will not be updated to use

CtorThisDerived's vtable until CtorThisDerived(). Therefore, virt_func() will call CtorThis::virt_func(), regardless of whether it is intended to call that or CtorThisDerived::virt_func().

Section 32.2: Using the this Pointer to Access Member Data

In this context, using the this pointer isn't entirely necessary, but it will make your code clearer to the reader, by indicating that a given function or variable is a member of the class. An example in this situation:

```
// Example for this pointer
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
  public:
    Class();
    ~Class();
    int getPrivateNumber () const;
  private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

See it in action here.

Section 32.3: Using the this Pointer to Differentiate Between Member Data and Parameters

This is an actual useful strategy to differentiate member data from parameters... Lets take this example :

```
// Dog Class Example
#include <iostream>
#include <string>

using std::cout;
using std::endl;

/*
* @class Dog
*   @member name
*       Dog's name
*   @function bark
*       Dog Barks!
*   @function getName
*       To Get Private
*       Name Variable
*/
class Dog
{
 public:
    Dog(std::string name);
    ~Dog();
    void  bark() const;
    std::string  getName() const;
 private:
    std::string name;
};

Dog::Dog(std::string name)
{
    /*
    *  this->name is the
    *  name variable from
    *  the class dog . and
    *  name is from the
    *  parameter of the function
    */
    this->name = name;
}

Dog::~Dog(){}

void Dog::bark() const
{
  cout << "BARK" << endl;
}

std::string  Dog::getName() const
{
    return this->name;
}


int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}
```

You can see here in the constructor we execute the following:

```
this->name = name;
```

Here , you can see we are assinging the parameter name to the name of the private variable from the class Dog(this->name) .

To see the output of above code : http://cpp.sh/75r7

Section 32.4: this Pointer CV-Qualifiers

`this` can also be cv-qualified, the same as any other pointer. However, due to the `this` parameter not being listed in the parameter list, special syntax is required for this; the cv-qualifiers are listed after the parameter list, but before the function's body.

```
struct ThisCVQ {
    void no_qualifier()                {} // "this" is: ThisCVQ*
    void  c_qualifier() const          {} // "this" is: const ThisCVQ*
    void  v_qualifier() volatile       {} // "this" is: volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is: const volatile ThisCVQ*
};
```

As `this` is a parameter, a function can be overloaded based on its `this` cv-qualifier(s).

```
struct CVOverload {
    int func()                { return    3; }
    int func() const          { return   33; }
    int func() volatile       { return  333; }
    int func() const volatile { return 3333; }
};
```

When `this` is `const` (including `const volatile`), the function is unable to write to member variables through it, whether implicitly or explicitly. The sole exception to this is `mutable` member variables, which can be written regardless of const-ness. Due to this, `const` is used to indicate that the member function doesn't change the object's logical state (the way the object appears to the outside world), even if it does modify the physical state (the way the object looks under the hood).

> Logical state is the way the object appears to outside observers. It isn't directly tied to physical state, and indeed, might not even be stored as physical state. As long as outside observers can't see any changes, the logical state is constant, even if you flip every single bit in the object.

> Physical state, also known as bitwise state, is how the object is stored in memory. This is the object's nitty-gritty, the raw 1s and 0s that make up its data. An object is only physically constant if its representation in memory never changes.

Note that C++ bases `const`ness on logical state, not physical state.

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somehow(const Param& p);

    // ...

  public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somehow(p);
    }

    void change_state(Param p) {
        modify_somehow(p);
        state_changed = true;
    }
    // Return some complex and/or expensive-to-calculate result.
    // As this has no reason to modify logical state, it is marked as "const".
    ResultType get_result() const;
};
ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result and state_changed can be modified, even with a const "this" pointer.
    // Even though the function doesn't modify logical state, it does modify physical state
    //  by caching the result, so it doesn't need to be recalculated every time the function
    //  is called.  This is indicated by cached_result and state_changed being mutable.

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}
```

Note that while you technically *could* use `const_cast` on `this` to make it non-cv-qualified, you really, ***REALLY*** shouldn't, and should use `mutable` instead. A `const_cast` is liable to invoke undefined behaviour when used on an object that actually *is* `const`, while `mutable` is designed to be safe to use. It is, however, possible that you may run into this in extremely old code.

An exception to this rule is defining non-cv-qualified accessors in terms of `const` accessors; as the object is guaranteed to not be `const` if the non-cv-qualified version is called, there's no risk of UB.

```
class CVAccessor {
    int arr[5];

  public:
    const int& get_arr_element(size_t i) const { return arr[i]; }

    int& get_arr_element(size_t i) {
        return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
    }
};
```

This prevents unnecessary duplication of code.

As with regular pointers, if `this` is `volatile` (including `const volatile`), it is loaded from memory each time it is accessed, instead of being cached. This has the same effects on optimisation as declaring any other pointer `volatile` would, so care should be taken.

Note that if an instance is cv-qualified, the only member functions it is allowed to access are member functions whose `this` pointer is at least as cv-qualified as the instance itself:

- Non-cv instances can access any member functions.
- `const` instances can access `const` and `const volatile` functions.
- `volatile` instances can access `volatile` and `const volatile` functions.
- `const volatile` instances can access `const volatile` functions.

This is one of the key tenets of `const` correctness.

```
struct CVAccess {
    void    func()              {}
    void  func_c() const        {}
    void  func_v() volatile     {}
    void func_cv() const volatile {}
};

CVAccess cva;
cva.func();    // Good.
cva.func_c();  // Good.
cva.func_v();  // Good.
cva.func_cv(); // Good.

const CVAccess c_cva;
c_cva.func();    // Error.
c_cva.func_c();  // Good.
c_cva.func_v();  // Error.
c_cva.func_cv(); // Good.

volatile CVAccess v_cva;
v_cva.func();    // Error.
v_cva.func_c();  // Error.
v_cva.func_v();  // Good.
v_cva.func_cv(); // Good.

const volatile CVAccess cv_cva;
cv_cva.func();    // Error.
cv_cva.func_c();  // Error.
cv_cva.func_v();  // Error.
cv_cva.func_cv(); // Good.
```

Section 32.5: this Pointer Ref-Qualifiers

`Version ≥ C++11`

Similarly to `this` cv-qualifiers, we can also apply *ref-qualifiers* to `*this`. Ref-qualifiers are used to choose between normal and rvalue reference semantics, allowing the compiler to use either copy or move semantics depending on which are more appropriate, and are applied to `*this` instead of `this`.

Note that despite ref-qualifiers using reference syntax, `this` itself is still a pointer. Also note that ref-qualifiers don't actually change the type of `*this`; it's just easier to describe and understand their effects by looking at them as if they did.

```
struct RefQualifiers {
    std::string s;

    RefQualifiers(const std::string& ss = "The nameless one.") : s(ss) {}

    // Normal version.
    void func() &  { std::cout << "Accessed on normal instance "    << s << std::endl; }
    // Rvalue version.
    void func() && { std::cout << "Accessed on temporary instance " << s << std::endl; }

    const std::string& still_a_pointer() &  { return this->s; }
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }
};

// ...

RefQualifiers rf("Fred");
rf.func();              // Output:  Accessed on normal instance Fred
RefQualifiers{}.func(); // Output:  Accessed on temporary instance The nameless one
```

A member function cannot have overloads both with and without ref-qualifiers; the programmer has to choose between one or the other. Thankfully, cv-qualifiers can be used in conjunction with ref-qualifiers, allowing `const` correctness rules to be followed.

```
struct RefCV {
    void func() &                {}
    void func() &&               {}
    void func() const&           {}
    void func() const&&          {}
    void func() volatile&        {}
    void func() volatile&&       {}
    void func() const volatile&  {}
    void func() const volatile&& {}
};
```

Chapter 33: Smart Pointers

Section 33.1: Unique ownership (std::unique_ptr)

Version ≥ C++11

A `std::unique_ptr` is a class template that manages the lifetime of a dynamically stored object. Unlike for `std::shared_ptr`, the dynamic object is owned by only *one instance* of a `std::unique_ptr` at any time,

```
// Creates a dynamic int with value of 20 owned by a unique pointer
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(Note: `std::unique_ptr` is available since C++11 and `std::make_unique` since C++14.)

Only the variable `ptr` holds a pointer to a dynamically allocated `int`. When a unique pointer that owns an object goes out of scope, the owned object is deleted, i.e. its destructor is called if the object is of class type, and the memory for that object is released.

To use `std::unique_ptr` and `std::make_unique` with array-types, use their array specializations:

```
// Creates a unique_ptr to an int with value 59
std::unique_ptr<int> ptr = std::make_unique<int>(59);

// Creates a unique_ptr to an array of 15 ints
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

You can access the `std::unique_ptr` just like a raw pointer, because it overloads those operators.

You can transfer ownership of the contents of a smart pointer to another pointer by using `std::move`, which will cause the original smart pointer to point to `nullptr`.

```
// 1. std::unique_ptr
std::unique_ptr<int> ptr = std::make_unique<int>();

// Change value to 1
*ptr = 1;

// 2. std::unique_ptr (by moving 'ptr' to 'ptr2', 'ptr' doesn't own the object anymore)
std::unique_ptr<int> ptr2 = std::move(ptr);

int a = *ptr2; // 'a' is 1
int b = *ptr;  // undefined behavior! 'ptr' is 'nullptr'
               // (because of the move command above)
```

Passing `unique_ptr` to functions as parameter:

```
void foo(std::unique_ptr<int> ptr)
{
    // Your code goes here
}

std::unique_ptr<int> ptr = std::make_unique<int>(59);
foo(std::move(ptr))
```

Returning `unique_ptr` from functions. This is the preferred C++11 way of writing factory functions, as it clearly conveys the ownership semantics of the return: the caller owns the resulting `unique_ptr` and is responsible for it.

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

Compare this to:

```
int* foo_cpp03();

int* p = foo_cpp03(); // do I own p? do I have to delete it at some point?
                      // it's not readily apparent what the answer is.
```

Version < C++14

The class template `make_unique` is provided since C++14. It's easy to add it manually to C++11 code:

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args)...)); }

// Use make_unique for arrays
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]()); }
```

Version ≥ C++11

Unlike the *dumb* smart pointer (`std::auto_ptr`), `unique_ptr` can also be instantiated with vector allocation (*not* `std::vector`). Earlier examples were for *scalar* allocations. For example to have a dynamically allocated integer array for 10 elements, you would specify `int[]` as the template type (and not just `int`):

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

Which can be simplified with:

```
auto arr_ptr = std::make_unique<int[]>(10);
```

Now, you use `arr_ptr` as if it is an array:

```
arr_ptr[2] =  10; // Modify third element
```

You need not to worry about de-allocation. This template specialized version calls constructors and destructors appropriately. Using vectored version of `unique_ptr` or a `vector` itself - is a personal choice.

In versions prior to C++11, `std::auto_ptr` was available. Unlike `unique_ptr` it is allowed to copy `auto_ptr`s, upon which the source `ptr` will lose the ownership of the contained pointer and the target receives it.

Section 33.2: Sharing ownership (std::shared_ptr)

The class template `std::shared_ptr` defines a shared pointer that is able to share ownership of an object with other shared pointers. This contrasts to `std::unique_ptr` which represents exclusive ownership.

The sharing behavior is implemented through a technique known as reference counting, where the number of shared pointers that point to the object is stored alongside it. When this count reaches zero, either through the destruction or reassignment of the last `std::shared_ptr` instance, the object is automatically destroyed.

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

To create multiple smart pointers that share the same object, we need to create another `shared_ptr` that aliases the first shared pointer. Here are 2 ways of doing it:

```
std::shared_ptr<Foo> secondShared(firstShared);  // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared;                      // 2nd way: Assigning
```

Either of the above ways makes `secondShared` a shared pointer that shares ownership of our instance of `Foo` with `firstShared`.

The smart pointer works just like a raw pointer. This means, you can use `*` to dereference them. The regular `->` operator works as well:

```
secondShared->test(); // Calls Foo::test()
```

Finally, when the last aliased `shared_ptr` goes out of scope, the destructor of our `Foo` instance is called.

Warning: Constructing a `shared_ptr` might throw a `bad_alloc` exception when extra data for shared ownership semantics needs to be allocated. If the constructor is passed a regular pointer it assumes to own the object pointed to and calls the deleter if an exception is thrown. This means `shared_ptr<T>(new T(args))` will not leak a `T` object if allocation of `shared_ptr<T>` fails. However, it is advisable to use `make_shared<T>(args)` or `allocate_shared<T>(alloc, args)`, which enable the implementation to optimize the memory allocation.

Allocating Arrays([]) using shared_ptr

Version ≥ C++11 Version < C++17

Unfortunately, there is no direct way to allocate Arrays using `make_shared<>`.

It is possible to create arrays for `shared_ptr<>` using `new` and `std::default_delete`.

For example, to allocate an array of 10 integers, we can write the code as

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

Specifying `std::default_delete` is mandatory here to make sure that the allocated memory is correctly cleaned up using `delete[]`.

If we know the size at compile time, we can do it this way:

```
template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
  std::shared_ptr<T> operator()const{
    auto r = std::make_shared<std::array<T,N>>();
    if (!r) return {};
    return {r.data(), r};
  }
};
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }
```

then `make_shared_array<int[10]>` returns a `shared_ptr<int>` pointing to 10 ints all default constructed.

Version ≥ C++17

With C++17, `shared_ptr` gained special support for array types. It is no longer necessary to specify the array-deleter explicitly, and the shared pointer can be dereferenced using the `[]` array index operator:

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

Shared pointers can point to a sub-object of the object it owns:

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

Both `p2` and `p1` own the object of type `Foo`, but `p2` points to its `int` member `x`. This means that if `p1` goes out of scope or is reassigned, the underlying `Foo` object will still be alive, ensuring that `p2` does not dangle.

Important: A `shared_ptr` only knows about itself and all other `shared_ptr` that were created with the alias constructor. It does not know about any other pointers, including all other `shared_ptr`s created with a reference to the same `Foo` instance:

```
Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // don't do this

shared1.reset(); // this will delete foo, since shared1
                 // was the only shared_ptr that owned it

shared2->test(); // UNDEFINED BEHAVIOR: shared2's foo has been
                 // deleted already!!
```

Ownership Transfer of shared_ptr

By default, `shared_ptr` increments the reference count and doesn't transfer the ownership. However, it can be made to transfer the ownership using `std::move`:

```
shared_ptr<int> up = make_shared<int>();
// Transferring the ownership
shared_ptr<int> up2 = move(up);
// At this point, the reference count of up = 0 and the
// ownership of the pointer is solely with up2 with reference count = 1
```

Section 33.3: Sharing with temporary ownership

(std::weak_ptr)

Instances of `std::weak_ptr` can point to objects owned by instances of `std::shared_ptr` while only becoming temporary owners themselves. This means that weak pointers do not alter the object's reference count and therefore do not prevent an object's deletion if all of the object's shared pointers are reassigned or destroyed.

In the following example instances of `std::weak_ptr` are used so that the destruction of a tree object is not inhibited:

```
#include <memory>
#include <vector>

struct TreeNode {
    std::weak_ptr<TreeNode> parent;
    std::vector< std::shared_ptr<TreeNode> > children;
};

int main() {
    // Create a TreeNode to serve as the root/parent.
    std::shared_ptr<TreeNode> root(new TreeNode);

    // Give the parent 100 child nodes.
    for (size_t i = 0; i < 100; ++i) {
        std::shared_ptr<TreeNode> child(new TreeNode);
        root->children.push_back(child);
        child->parent = root;
    }

    // Reset the root shared pointer, destroying the root object, and
    // subsequently its child nodes.
    root.reset();
}
```

As child nodes are added to the root node's children, their `std::weak_ptr` member `parent` is set to the root node. The member `parent` is declared as a weak pointer as opposed to a shared pointer such that the root node's reference count is not incremented. When the root node is reset at the end of `main()`, the root is destroyed. Since the only remaining `std::shared_ptr` references to the child nodes were contained in the root's collection `children`, all child nodes are subsequently destroyed as well.

Due to control block implementation details, shared_ptr allocated memory may not be released until `shared_ptr` reference counter and `weak_ptr` reference counter both reach zero.

```
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         {
             // std::make_shared is optimized by allocating only once
             // while std::shared_ptr<int>(new int(42)) allocates twice.
             // Drawback of std::make_shared is that control block is tied to our integer
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // sh memory should be released at this point...
         }
         // ... but wk is still alive and needs access to control block
     }
     // now memory is released (sh and wk)
}
```

Since `std::weak_ptr` does not keep its referenced object alive, direct data access through a `std::weak_ptr` is not possible. Instead it provides a `lock()` member function that attempts to retrieve a `std::shared_ptr` to the referenced object:

```
#include <cassert>
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         std::shared_ptr<int> sp;
         {
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // calling lock will create a shared_ptr to the object referenced by wk
             sp = wk.lock();
             // sh will be destroyed after this point, but sp is still alive
         }
         // sp still keeps the data alive.
         // At this point we could even call lock() again
         // to retrieve another shared_ptr to the same data from wk
         assert(*sp == 42);
         assert(!wk.expired());
         // resetting sp will delete the data,
         // as it is currently the last shared_ptr with ownership
         sp.reset();
         // attempting to lock wk now will return an empty shared_ptr,
         // as the data has already been deleted
         sp = wk.lock();
         assert(!sp);
         assert(wk.expired());
     }
}
```

Section 33.4: Using custom deleters to create a wrapper to a C interface

Many C interfaces such as SDL2 have their own deletion functions. This means that you cannot use smart pointers directly:

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

Instead, you need to define your own deleter. The examples here use the `SDL_Surface` structure which should be freed using the `SDL_FreeSurface()` function, but they should be adaptable to many other C interfaces.

The deleter must be callable with a pointer argument, and therefore can be e.g. a simple function pointer:

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

Any other callable object will work, too, for example a class with an `operator()`:

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};
```

```
std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // safe
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // equivalent to the above
                                                         // as the deleter is value-initialized
```

This not only provides you with safe, zero overhead (if you use unique_ptr) automatic memory management, you also get exception safety.

Note that the deleter is part of the type for unique_ptr, and the implementation can use the empty base optimization to avoid any change in size for empty custom deleters. So while std::unique_ptr<SDL_Surface, SurfaceDeleter> and std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> solve the same problem in a similar way, the former type is still only the size of a pointer while the latter type has to hold *two* pointers: both the SDL_Surface* and the function pointer! When having free function custom deleters, it is preferable to wrap the function in an empty type.

In cases where reference counting is important, one could use a shared_ptr instead of an unique_ptr. The shared_ptr always stores a deleter, this erases the type of the deleter, which might be useful in APIs. The disadvantages of using shared_ptr over unique_ptr include a higher memory cost for storing the deleter and a performance cost for maintaining the reference count.

```
// deleter required at construction time and is part of the type
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);

// deleter is only required at construction time, not part of the type
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);
```

Version ≥ C++17

With template auto, we can make it even easier to wrap our custom deleters:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) {
        DeleteFn(ptr);
    }
};

template <class T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

With which the above example is simply:

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

Here, the purpose of auto is to handle all free functions, whether they return void (e.g. SDL_FreeSurface) or not (e.g. fclose).

Section 33.5: Unique ownership without move semantics (auto_ptr)

Version < C++11

NOTE: std::auto_ptr has been deprecated in C++11 and will be removed in C++17. You should only use this if you are forced to use C++03 or earlier and are willing to be careful. It is recommended to move to unique_ptr in combination with std::move to replace std::auto_ptr behavior.

Before we had std::unique_ptr, before we had move semantics, we had std::auto_ptr. std::auto_ptr provides unique ownership but transfers ownership upon copy.

As with all smart pointers, std::auto_ptr automatically cleans up resources (see RAII):

```
{
    std::auto_ptr<int> p(new int(42));
    std::cout << *p;
} // p is deleted here, no memory leaked
```

but allows only one owner:

```
std::auto_ptr<X> px = ...;
std::auto_ptr<X> py = px;
  // px is now empty
```

This allows to use std::auto_ptr to keep ownership explicit and unique at the danger of losing ownership unintended:

```
void f(std::auto_ptr<X> ) {
    // assumes ownership of X
    // deletes it at end of scope
};

std::auto_ptr<X> px = ...;
f(px); // f acquires ownership of underlying X
       // px is now empty
px->foo(); // NPE!
// px.~auto_ptr() does NOT delete
```

The transfer of ownership happened in the "copy" constructor. auto_ptr's copy constructor and copy assignment operator take their operands by non-const reference so that they could be modified. An example implementation might be:

```
template <typename T>
class auto_ptr {
    T* ptr;
public:
    auto_ptr(auto_ptr& rhs)
    : ptr(rhs.release())
    { }

    auto_ptr& operator=(auto_ptr& rhs) {
        reset(rhs.release());
        return *this;
    }

    T* release() {
        T* tmp = ptr;
        ptr = nullptr;
        return tmp;
    }

    void reset(T* tmp = nullptr) {
        if (ptr != tmp) {
            delete ptr;
            ptr = tmp;
        }
    }
    /* other functions ... */
};
```

This breaks copy semantics, which require that copying an object leaves you with two equivalent versions of it. For any copyable type, `T`, I should be able to write:

```
T a = ...;
T b(a);
assert(b == a);
```

But for `auto_ptr`, this is not the case. As a result, it is not safe to put `auto_ptr`s in containers.

Section 33.6: Casting std::shared_ptr pointers

It is not possible to directly use `static_cast`, `const_cast`, `dynamic_cast` and `reinterpret_cast` on `std::shared_ptr` to retrieve a pointer sharing ownership with the pointer being passed as argument. Instead, the functions `std::static_pointer_cast`, `std::const_pointer_cast`, `std::dynamic_pointer_cast` and `std::reinterpret_pointer_cast` should be used:

```
struct Base { virtual ~Base() noexcept {}; };
struct Derived: Base {};
auto derivedPtr(std::make_shared<Derived>());
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

Note that `std::reinterpret_pointer_cast` is not available in C++11 and C++14, as it was only proposed by N3920 and adopted into Library Fundamentals TS in February 2014. However, it can be implemented as follows:

```
template <typename To, typename From>
inline std::shared_ptr<To> reinterpret_pointer_cast(
    std::shared_ptr<From> const & ptr) noexcept
{ return std::shared_ptr<To>(ptr, reinterpret_cast<To *>(ptr.get())); }
```

Section 33.7: Writing a smart pointer: value_ptr

A `value_ptr` is a smart pointer that behaves like a value. When copied, it copies its contents. When created, it creates its contents.

```
// Like std::default_delete:
template<class T>
struct default_copier {
  // a copier must handle a null T const* in and return null:
  T* operator()(T const* tin)const {
    if (!tin) return nullptr;
    return new T(*tin);
  }
  void operator()(void* dest, T const* tin)const {
    if (!tin) return;
    return new(dest) T(*tin);
  }
};
// tag class to handle empty case:
struct empty_ptr_t {};
constexpr empty_ptr_t empty_ptr{};
// the value pointer type itself:
```

```
template<class T, class Copier=default_copier<T>, class Deleter=std::default_delete<T>,
  class Base=std::unique_ptr<T, Deleter>
>
struct value_ptr:Base, private Copier {
  using copier_type=Copier;
  // also typedefs from unique_ptr

  using Base::Base;

  value_ptr( T const& t ):
    Base( std::make_unique<T>(t) ),
    Copier()
  {}
  value_ptr( T && t ):
    Base( std::make_unique<T>(std::move(t)) ),
    Copier()
  {}
  // almost-never-empty:
      value_ptr():
    Base( std::make_unique<T>() ),
    Copier()
  {}
  value_ptr( empty_ptr_t ) {}

  value_ptr( Base b, Copier c={} ):
    Base(std::move(b)),
    Copier(std::move(c))
  {}

  Copier const& get_copier() const {
    return *this;
  }

  value_ptr clone() const {
    return {
      Base(
        get_copier()(this->get()),
        this->get_deleter()
      ),
      get_copier()
    };
  }
  value_ptr(value_ptr&&)=default;
  value_ptr& operator=(value_ptr&&)=default;

  value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
  value_ptr& operator=(value_ptr const&o) {
    if (o && *this) {
      // if we are both non-null, assign contents:
      **this = *o;
    } else {
      // otherwise, assign a clone (which could itself be null):
      *this = o.clone();
    }
    return *this;
  }
  value_ptr& operator=( T const& t ) {
    if (*this) {
      **this = t;
    } else {
      *this = value_ptr(t);
    }
    return *this;
  }
  value_ptr& operator=( T && t ) {
    if (*this) {
      **this = std::move(t);
    } else {
      *this = value_ptr(std::move(t));
    }
    return *this;
  }
  T& get() { return **this; }
  T const& get() const { return **this; }
  T* get_pointer() {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  // operator-> from unique_ptr
};
template<class T, class...Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
  return {std::make_unique<T>(std::forward<Args>(args)...)};
}
```

This particular value_ptr is only empty if you construct it with `empty_ptr_t` or if you move from it. It exposes the fact it is a `unique_ptr`, so `explicit operator bool() const` works on it. `.get()` has been changed to return a reference (as it is almost never empty), and `.get_pointer()` returns a pointer instead.

This smart pointer can be useful for `pImpl` cases, where we want value-semantics but we also don't want to expose the contents of the `pImpl` outside of the implementation file.

With a non-default `Copier`, it can even handle virtual base classes that know how to produce instances of their derived and turn them into value-types.

Section 33.8: Getting a shared_ptr referring to this

`enable_shared_from_this` enables you to get a valid `shared_ptr` instance to `this`.

By deriving your class from the class template `enable_shared_from_this`, you inherit a method `shared_from_this` that returns a `shared_ptr` instance to `this`.

Note that the object must be created as a `shared_ptr` in first place:

```
#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 =new A();
shared_ptr<A> ap2(ap1); // First prepare a shared pointer to the object and hold it!
// Then get a shared pointer to the object from the object itself
shared_ptr<A> ap3 = ap1->shared_from_this();
int c3 =ap3.use_count(); // =2: pointing to the same object
```

Note(2) you cannot call `enable_shared_from_this` inside the constructor.

```
#include <memory> // enable_shared_from_this
```

```
class Widget : public std::enable_shared_from_this< Widget >
{
public:
    void DoSomething()
    {
        std::shared_ptr< Widget > self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared< Widget >();
    w -> DoSomething();
    ...
}
```

If you use `shared_from_this()` on an object not owned by a `shared_ptr`, such as a local automatic object or a global object, then the behavior is undefined. Since C++17 it throws `std::bad_alloc` instead.

Using `shared_from_this()` from a constructor is equivalent to using it on an object not owned by a `shared_ptr`, because the objects is possessed by the `shared_ptr` after the constructor returns.

Chapter 34: Classes/Structures

Section 34.1: Class basics

A *class* is a user-defined type. A class is introduced with the `class`, `struct` or `union` keyword. In colloquial usage, the term "class" usually refers only to non-union classes.

A class is a collection of *class members*, which can be:

- member variables (also called "fields"),
- member functions (also called "methods"),
- member types or typedefs (e.g. "nested classes"),
- member templates (of any kind: variable, function, class or alias template)

The `class` and `struct` keywords, called *class keys*, are largely interchangeable, except that the default access specifier for members and bases is "private" for a class declared with the `class` key and "public" for a class declared with the `struct` or `union` key (cf. Access modifiers).

For example, the following code snippets are identical:

```
struct Vector
{
    int x;
    int y;
    int z;
};
// are equivalent to
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

By declaring a class` a new type is added to your program, and it is possible to instantiate objects of that class by

```
Vector my_vector;
```

Members of a class are accessed using dot-syntax.

```
my_vector.x = 10;
my_vector.y = my_vector.x + 1; // my_vector.y = 11;
my_vector.z = my_vector.y - 4; // my:vector.z = 7;
```

Section 34.2: Final classes and structs

Version ≥ C++11

Deriving a class may be forbidden with `final` specifier. Let's declare a final class:

```
class A final {
};
```

Now any attempt to subclass it will cause a compilation error:

```
// Compilation error: cannot derive from final class:
class B : public A {
};
```

Final class may appear anywhere in class hierarchy:

```
class A {
};

// OK.
class B final : public A {
};

// Compilation error: cannot derive from final class B.
class C : public B {
};
```

Section 34.3: Access specifiers

There are three keywords that act as **access specifiers**. These limit the access to class members following the specifier, until another specifier changes the access level again:

Keyword	Description
`public`	Everyone has access
`protected`	Only the class itself, derived classes and friends have access
`private`	Only the class itself and friends have access

When the type is defined using the `class` keyword, the default access specifier is `private`, but if the type is defined using the `struct` keyword, the default access specifier is `public`:

```
struct MyStruct { int x; };
class MyClass { int x; };

MyStruct s;
s.x = 9; // well formed, because x is public

MyClass c;
c.x = 9; // ill-formed, because x is private
```

Access specifiers are mostly used to limit access to internal fields and methods, and force the programmer to use a specific interface, for example to force use of getters and setters instead of referencing a variable directly:

```
class MyClass {

public: /* Methods: */

    int x() const noexcept { return m_x; }
    void setX(int const x) noexcept { m_x = x; }

private: /* Fields: */

    int m_x;

};
```

Using `protected` is useful for allowing certain functionality of the type to be only accessible to the derived classes, for example, in the following code, the method `calculateValue()` is only accessible to classes deriving from the base class Plus2Base, such as FortyTwo:

```
struct Plus2Base {
    int value() noexcept { return calculateValue() + 2; }
protected: /* Methods: */
    virtual int calculateValue() noexcept = 0;
};
struct FortyTwo: Plus2Base {
protected: /* Methods: */
    int calculateValue() noexcept final override { return 40; }
};
```

Note that the friend keyword can be used to add access exceptions to functions or types for accessing protected and private members.

The public, protected, and private keywords can also be used to grant or limit access to base class subobjects. See the Inheritance example.

Section 34.4: Inheritance

Classes/structs can have inheritance relations.

If a class/struct B inherits from a class/struct A, this means that B has as a parent A. We say that B is a derived class/struct from A, and A is the base class/struct.

```
struct A
{
public:
    int p1;
protected:
    int p2;
private:
    int p3;
};

//Make B inherit publicly (default) from A
struct B : A
{
};
```

There are 3 forms of inheritance for a class/struct:

- public
- private
- protected

Note that the default inheritance is the same as the default visibility of members: public if you use the struct keyword, and private for the class keyword.

It's even possible to have a class derive from a struct (or vice versa). In this case, the default inheritance is controlled by the child, so a struct that derives from a class will default to public inheritance, and a class that derives from a struct will have private inheritance by default.

public inheritance:

```
struct B : public A // or just `struct B : A`
{
    void foo()
    {
        p1 = 0; //well formed, p1 is public in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //well formed, p1 is public
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

`private` inheritance:

```
struct B : private A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is private in B
        p2 = 0; //well formed, p2 is private in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is private
b.p2 = 1; //ill formed, p2 is private
b.p3 = 1; //ill formed, p3 is inaccessible
```

`protected` inheritance:

```
struct B : protected A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is protected in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is protected
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

Note that although `protected` inheritance is allowed, the actual use of it is rare. One instance of how `protected` inheritance is used in application is in partial base class specialization (usually referred to as "controlled polymorphism").

When OOP was relatively new, (public) inheritance was frequently said to model an "IS-A" relationship. That is, public inheritance is correct only if an instance of the derived class *is also an* instance of the base class.

This was later refined into the Liskov Substitution Principle: public inheritance should only be used when/if an instance of the derived class can be substituted for an instance of the base class under any possible circumstance (and still make sense).

Private inheritance is typically said to embody a completely different relationship: "is implemented in terms of" (sometimes called a "HAS-A" relationship). For example, a Stack class could inherit privately from a Vector class. Private inheritance bears a much greater similarity to aggregation than to public inheritance.

Protected inheritance is almost never used, and there's no general agreement on what sort of relationship it embodies.

Section 34.5: Friendship

The friend keyword is used to give other classes and functions access to private and protected members of the class, even through they are defined outside the class`s scope.

```
class Animal{
private:
    double weight;
    double height;
public:
    friend void printWeight(Animal animal);
    friend class AnimalPrinter;
    // A common use for a friend function is to overload the operator<< for streaming.
    friend std::ostream& operator<<(std::ostream& os, Animal animal);
};

void printWeight(Animal animal)
{
    std::cout << animal.weight << "\n";
}

class AnimalPrinter
{
public:
    void print(const Animal& animal)
    {
        // Because of the `friend class AnimalPrinter;" declaration, we are
        // allowed to access private members here.
        std::cout << animal.weight << ", " << animal.height << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, Animal animal)
{
    os << "Animal height: " << animal.height << "\n";
    return os;
}

int main() {
    Animal animal = {10, 5};
    printWeight(animal);

    AnimalPrinter aPrinter;
    aPrinter.print(animal);

    std::cout << animal;
}
```

```
10
10, 5
Animal height: 5
```

Section 34.6: Virtual Inheritance

When using inheritance, you can specify the `virtual` keyword:

```
struct A{};
struct B: public virtual A{};
```

When class `B` has virtual base `A` it means that `A` **will reside in most derived class** of inheritance tree, and thus that most derived class is also responsible for initializing that virtual base:

```
struct A
{
    int member;
    A(int param)
    {
        member = param;
    }
};

struct B: virtual A
{
    B(): A(5){}
};

struct C: B
{
    C(): /*A(88)*/ {}
};

void f()
{
    C object; //error since C is not initializing it's indirect virtual base `A`
}
```

If we un-comment `/*A(88)*/` we won't get any error since `C` is now initializing it's indirect virtual base `A`.

Also note that when we're creating variable `object`, most derived class is `C`, so `C` is responsible for creating(calling constructor of) `A` and thus value of `A::member` is `88`, not `5` (as it would be if we were creating object of type `B`).

It is useful when solving the diamond problem.:

```
  A                                        A   A
 / \                                       |   |
B   C                                      B   C
 \ /                                        \ /
  D                                          D
virtual inheritance                   normal inheritance
```

`B` and `C` both inherit from `A`, and `D` inherits from `B` and `C`, so **there are 2 instances of `A` in `D`!** This results in ambiguity when you're accessing member of `A` through `D`, as the compiler has no way of knowing from which class do you want to access that member (the one which `B` inherits, or the one that is inherited by`C`?).

Virtual inheritance solves this problem: Since virtual base resides only in most derived object, there will be only one instance of `A` in `D`.

```
struct A
{
    void foo() {}
};

struct B : public /*virtual*/ A {};
struct C : public /*virtual*/ A {};

struct D : public B, public C
{
    void bar()
    {
        foo(); //Error, which foo? B::foo() or C::foo()? - Ambiguous
    }
};
```

Removing the comments resolves the ambiguity.

Section 34.7: Private inheritance: restricting base class interface

Private inheritance is useful when it is required to restrict the public interface of the class:

```
class A {
public:
    int move();
    int turn();
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move();  // compile error
b.turn();  // OK
```

This approach efficiently prevents an access to the A public methods by casting to the A pointer or reference:

```
B b;
A& a = static_cast<A&>(b); // compile error
```

In the case of public inheritance such casting will provide access to all the A public methods despite on alternative ways to prevent this in derived B, like hiding:

```
class B : public A {
private:
    int move();
};
```

or private using:

```
class B : public A {
private:
    using A::move;
};
```

then for both cases it is possible:

```
B b;
```

```
A& a = static_cast<A&>(b); // OK for public inheritance
a.move(); // OK
```

Section 34.8: Accessing class members

To access member variables and member functions of an object of a class, the `.` operator is used:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
// Accessing member variable a in var.
std::cout << var.a << std::endl;
// Assigning member variable b in var.
var.b = 1;
// Calling a member function.
var.foo();
```

When accessing the members of a class via a pointer, the `->` operator is commonly used. Alternatively, the instance can be dereferenced and the `.` operator used, although this is less common:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
SomeStruct *p = &var;
// Accessing member variable a in var via pointer.
std::cout << p->a << std::endl;
std::cout << (*p).a << std::endl;
// Assigning member variable b in var via pointer.
p->b = 1;
(*p).b = 1;
// Calling a member function via a pointer.
p->foo();
(*p).foo();
```

When accessing static class members, the `::` operator is used, but on the name of the class instead of an instance of it. Alternatively, the static member can be accessed from an instance or a pointer to an instance using the `.` or `->` operator, respectively, with the same syntax as accessing non-static members.

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}

  static int c;
  static void bar() {}
};
int SomeStruct::c;

SomeStruct var;
SomeStruct* p = &var;
// Assigning static member variable c in struct SomeStruct.
```

```
SomeStruct::c = 5;
// Accessing static member variable c in struct SomeStruct, through var and p.
var.a = var.c;
var.b = p->c;
// Calling a static member function.
SomeStruct::bar();
var.bar();
p->bar();
```

Background

The `->` operator is needed because the member access operator `.` has precedence over the dereferencing operator `*`.

One would expect that `*p.a` would dereference `p` (resulting in a reference to the object `p` is pointing to) and then accessing its member `a`. But in fact, it tries to access the member `a` of `p` and then dereference it. I.e. `*p.a` is equivalent to `*(p.a)`. In the example above, this would result in a compiler error because of two facts: First, `p` is a pointer and does not have a member `a`. Second, `a` is an integer and, thus, can't be dereferenced.

The uncommonly used solution to this problem would be to explicitly control the precedence: `(*p).a`

Instead, the `->` operator is almost always used. It is a short-hand for first dereferencing the pointer and then accessing it. I.e. `(*p).a` is exactly the same as `p->a`.

The `::` operator is the scope operator, used in the same manner as accessing a member of a namespace. This is because a static class member is considered to be in that class' scope, but isn't considered a member of instances of that class. The use of normal `.` and `->` is also allowed for static members, despite them not being instance members, for historical reasons; this is of use for writing generic code in templates, as the caller doesn't need to be concerned with whether a given member function is static or non-static.

Section 34.9: Member Types and Aliases

A `class` or `struct` can also define member type aliases, which are type aliases contained within, and treated as members of, the class itself.

```
struct IHaveATypedef {
    typedef int MyTypedef;
};

struct IHaveATemplateTypedef {
    template<typename T>
    using MyTemplateTypedef = std::vector<T>;
};
```

Like static members, these typedefs are accessed using the scope operator, `::`.

```
IHaveATypedef::MyTypedef i = 5; // i is an int.

IHaveATemplateTypedef::MyTemplateTypedef<int> v; // v is a std::vector<int>.
```

As with normal type aliases, each member type alias is allowed to refer to any type defined or aliased before, but not after, its definition. Likewise, a typedef outside the class definition can refer to any accessible typedefs within the class definition, provided it comes after the class definition.

```
template<typename T>
struct Helper {
    T get() const { return static_cast<T>(42); }
};

struct IHaveTypedefs {
//    typedef MyTypedef NonLinearTypedef; // Error if uncommented.
    typedef int MyTypedef;
    typedef Helper<MyTypedef> MyTypedefHelper;
};

IHaveTypedefs::MyTypedef        i; // x_i is an int.
IHaveTypedefs::MyTypedefHelper hi; // x_hi is a Helper<int>.

typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii;                  // ii is an int.
```

Member type aliases can be declared with any access level, and will respect the appropriate access modifier.

```
class TypedefAccessLevels {
    typedef int PrvInt;

  protected:
    typedef int ProInt;

  public:
    typedef int PubInt;
};

TypedefAccessLevels::PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
TypedefAccessLevels::ProInt pro_i; // Error: TypedefAccessLevels::ProInt is protected.
TypedefAccessLevels::PubInt pub_i; // Good.

class Derived : public TypedefAccessLevels {
    PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
    ProInt pro_i; // Good.
    PubInt pub_i; // Good.
};
```

This can be used to provide a level of abstraction, allowing a class' designer to change its internal workings without breaking code that relies on it.

```
class Something {
    friend class SomeComplexType;

    short s;
    // ...

  public:
    typedef SomeComplexType MyHelper;

    MyHelper get_helper() const { return MyHelper(8, s, 19.5, "shoe", false); }

    // ...
};

// ...

Something s;
Something::MyHelper hlp = s.get_helper();
```

In this situation, if the helper class is changed from `SomeComplexType` to some other type, only the `typedef` and the friend declaration would need to be modified; as long as the helper class provides the same functionality, any code that uses it as Something::MyHelper instead of specifying it by name will usually still work without any modifications. In this manner, we minimise the amount of code that needs to be modified when the underlying implementation is changed, such that the type name only needs to be changed in one location.

This can also be combined with decltype, if one so desires.

```
class SomethingElse {
    AnotherComplexType<bool, int, SomeThirdClass> helper;

  public:
    typedef decltype(helper) MyHelper;

  private:
    InternalVariable<MyHelper> ivh;

    // ...

  public:
    MyHelper& get_helper() const { return helper; }

    // ...
};
```

In this situation, changing the implementation of SomethingElse::helper will automatically change the typedef for us, due to decltype. This minimises the number of modifications necessary when we want to change helper, which minimises the risk of human error.

As with everything, however, this can be taken too far. If the typename is only used once or twice internally and zero times externally, for example, there's no need to provide an alias for it. If it's used hundreds or thousands of times throughout a project, or if it has a long enough name, then it can be useful to provide it as a typedef instead of always using it in absolute terms. One must balance forwards compatibility and convenience with the amount of unnecessary noise created.

This can also be used with template classes, to provide access to the template parameters from outside the class.

```
template<typename T>
class SomeClass {
    // ...

  public:
    typedef T MyParam;
    MyParam getParam() { return static_cast<T>(42); }
};

template<typename T>
typename T::MyParam some_func(T& t) {
    return t.getParam();
}

SomeClass<int> si;
int i = some_func(si);
```

This is commonly used with containers, which will usually provide their element type, and other helper types, as member type aliases. Most of the containers in the C++ standard library, for example, provide the following 12 helper types, along with any other special types they might need.

```
template<typename T>
```

```
class SomeContainer {
    // ...

  public:
    // Let's provide the same helper types as most standard containers.
    typedef T                                     value_type;
    typedef std::allocator<value_type>            allocator_type;
    typedef value_type&                           reference;
    typedef const value_type&                     const_reference;
    typedef value_type*                           pointer;
    typedef const value_type*                     const_pointer;
    typedef MyIterator<value_type>                iterator;
    typedef MyConstIterator<value_type>           const_iterator;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t                                size_type;
    typedef ptrdiff_t                             difference_type;
};
```

Prior to C++11, it was also commonly used to provide a "template `typedef`" of sorts, as the feature wasn't yet available; these have become a bit less common with the introduction of alias templates, but are still useful in some situations (and are combined with alias templates in other situations, which can be very useful for obtaining individual components of a complex type such as a function pointer). They commonly use the name `type` for their type alias.

```
template<typename T>
struct TemplateTypedef {
    typedef T type;
}

TemplateTypedef<int>::type i; // i is an int.
```

This was often used with types with multiple template parameters, to provide an alias that defines one or more of the parameters.

```
template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
};

OneDArray<int, 3>::type     arr1i; // arr1i is an Array<int, 3, 1>.
TwoDArray<short, 5>::type   arr2s; // arr2s is an Array<short, 5, 2>.
MonoDisplayLine<char>::type arr3c; // arr3c is an Array<char, 80, 1>.
```

Section 34.10: Nested Classes/Structures

A class or struct can also contain another class/struct definition inside itself, which is called a "nested class"; in this situation, the containing class is referred to as the "enclosing class". The nested class definition is considered to be a member of the enclosing class, but is otherwise separate.

```
struct Outer {
    struct Inner { };
};
```

From outside of the enclosing class, nested classes are accessed using the scope operator. From inside the enclosing class, however, nested classes can be used without qualifiers:

```
struct Outer {
    struct Inner { };

    Inner in;
};

// ...

Outer o;
Outer::Inner i = o.in;
```

As with a non-nested class/struct, member functions and static variables can be defined either within a nested class, or in the enclosing namespace. However, they cannot be defined within the enclosing class, due to it being considered to be a different class than the nested class.

```
// Bad.
struct Outer {
    struct Inner {
        void do_something();
    };

    void Inner::do_something() {}
};


// Good.
struct Outer {
    struct Inner {
        void do_something();
    };

};

void Outer::Inner::do_something() {}
```

As with non-nested classes, nested classes can be forward declared and defined later, provided they are defined before being used directly.

```
class Outer {
    class Inner1;
    class Inner2;

    class Inner1 {};

    Inner1 in1;
    Inner2* in2p;

  public:
    Outer();
    ~Outer();
};

class Outer::Inner2 {};

Outer::Outer() : in1(Inner1()), in2p(new Inner2) {}
Outer::~Outer() {
    if (in2p) { delete in2p; }
}
```

Version < C++11

Prior to C++11, nested classes only had access to type names, `static` members, and enumerators from the enclosing class; all other members defined in the enclosing class were off-limits.

Version ≥ C++11

As of C++11, nested classes, and members thereof, are treated as if they were `friend`s of the enclosing class, and can access all of its members, according to the usual access rules; if members of the nested class require the ability to evaluate one or more non-static members of the enclosing class, they must therefore be passed an instance:

```
class Outer {
    struct Inner {
        int get_sizeof_x() {
            return sizeof(x); // Legal (C++11): x is unevaluated, so no instance is required.
        }

        int get_x() {
            return x; // Illegal: Can't access non-static member without an instance.
        }

        int get_x(Outer& o) {
            return o.x; // Legal (C++11): As a member of Outer, Inner can access private members.
        }
    };

    int x;
};
```

Conversely, the enclosing class is *not* treated as a friend of the nested class, and thus cannot access its private members without explicitly being granted permission.

```
class Outer {
    class Inner {
        // friend class Outer;

        int x;
    };

    Inner in;

  public:
    int get_x() {
        return in.x; // Error: int Outer::Inner::x is private.
        // Uncomment "friend" line above to fix.
    }
};
```

Friends of a nested class are not automatically considered friends of the enclosing class; if they need to be friends of the enclosing class as well, this must be declared separately. Conversely, as the enclosing class is not automatically considered a friend of the nested class, neither will friends of the enclosing class be considered friends of the nested class.

```
class Outer {
    friend void barge_out(Outer& out, Inner& in);

    class Inner {
        friend void barge_in(Outer& out, Inner& in);

        int i;
    };

    int o;
};

void barge_in(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Good.
    int o = out.o; // Error: int Outer::o is private.
}

void barge_out(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Error: int Outer::Inner::i is private.
    int o = out.o; // Good.
}
```

As with all other class members, nested classes can only be named from outside the class if they have public access. However, you are allowed to access them regardless of access modifier, as long as you don't explicitly name them.

```
class Outer {
    struct Inner {
        void func() { std::cout << "I have no private taboo.\n"; }
    };

  public:
    static Inner make_Inner() { return Inner(); }
};

// ...

Outer::Inner oi; // Error: Outer::Inner is private.

auto oi = Outer::make_Inner(); // Good.
oi.func();                     // Good.
Outer::make_Inner().func();    // Good.
```

You can also create a type alias for a nested class. If a type alias is contained in the enclosing class, the nested type and the type alias can have different access modifiers. If the type alias is outside the enclosing class, it requires that either the nested class, or a `typedef` thereof, be public.

```
class Outer {
    class Inner_ {};

  public:
    typedef Inner_ Inner;
};

typedef Outer::Inner  ImOut; // Good.
typedef Outer::Inner_ ImBad; // Error.

// ...

Outer::Inner  oi; // Good.
Outer::Inner_ oi; // Error.
ImOut         oi; // Good.
```

As with other classes, nested classes can both derive from or be derived from by other classes.

```
struct Base {};

struct Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};
```

This can be useful in situations where the enclosing class is derived from by another class, by allowing the programmer to update the nested class as necessary. This can be combined with a typedef to provide a consistent name for each enclosing class' nested class:

```
class BaseOuter {
    struct BaseInner_ {
        virtual void do_something() {}
        virtual void do_something_else();
    } b_in;

  public:
    typedef BaseInner_ Inner;

    virtual ~BaseOuter() = default;

    virtual Inner& getInner() { return b_in; }
};

void BaseOuter::BaseInner_::do_something_else() {}

// ---

class DerivedOuter : public BaseOuter {
    // Note the use of the qualified typedef; BaseOuter::BaseInner_ is private.
    struct DerivedInner_ : BaseOuter::Inner {
        void do_something() override {}
        void do_something_else() override;
    } d_in;

  public:
    typedef DerivedInner_ Inner;

    BaseOuter::Inner& getInner() override { return d_in; }
};

void DerivedOuter::DerivedInner_::do_something_else() {}

// ...
// Calls BaseOuter::BaseInner_::do_something();
BaseOuter* b = new BaseOuter;
BaseOuter::Inner& bin = b->getInner();
bin.do_something();
b->getInner().do_something();

// Calls DerivedOuter::DerivedInner_::do_something();
BaseOuter* d = new DerivedOuter;
BaseOuter::Inner& din = d->getInner();
din.do_something();
d->getInner().do_something();
```

In the above case, both `BaseOuter` and `DerivedOuter` supply the member type `Inner`, as `BaseInner_` and `DerivedInner_`, respectively. This allows nested types to be derived without breaking the enclosing class' interface, and allows the nested type to be used polymorphically.

Section 34.11: Unnamed struct/class

Unnamed `struct` is allowed (type has no name)

```
void foo()
{
    struct /* No name */ {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

or

```
struct Circle
{
    struct /* No name */ {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

and later

```
Circle circle;
circle.center.x = 42.f;
```

but NOT *anonymous* `struct` (unnamed type and unnamed object)

```
struct InvalidCircle
{
    struct /* No name */ {
        float centerX;
        float centerY;
    }; // No member either.
    float radius;
};
```

Note: Some compilers allow *anonymous* `struct` as *extension*.

Version ≥ C++11

- *lamdba* can be seen as a special *unnamed* `struct`.
- `decltype` allows to retrieve the type of *unnamed* `struct`:

```
decltype(circle.point) otherPoint;
```

- *unnamed* `struct` instance can be parameter of template method:

```
void print_square_coordinates()
{
    const struct {float x; float y;} points[] = {
        {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
    };

    // for range relies on `template <class T, std::size_t N> std::begin(T (&)[N])`
    for (const auto& point : points) {
        std::cout << "{" << point.x << ", " << point.y << "}\n";
    }

    decltype(points[0]) topRightCorner{1, 1};
    auto it = std::find(points, points + 4, topRightCorner);
    std::cout << "top right corner is the "
              << 1 + std::distance(points, it) << "th\n";
}
```

Section 34.12: Static class members

A class is also allowed to have `static` members, which can be either variables or functions. These are considered to be in the class' scope, but aren't treated as normal members; they have static storage duration (they exist from the start of the program to the end), aren't tied to a particular instance of the class, and only one copy exists for the entire class.

```
class Example {
    static int num_instances;      // Static data member (static member variable).
    int i;                         // Non-static member variable.

  public:
    static std::string static_str; // Static data member (static member variable).
    static int static_func();      // Static member function.

    // Non-static member functions can modify static member variables.
    Example() { ++num_instances; }
    void set_str(const std::string& str);
};

int         Example::num_instances;
std::string Example::static_str = "Hello.";

// ...

Example one, two, three;
// Each Example has its own "i", such that:
//  (&one.i != &two.i)
//  (&one.i != &three.i)
//  (&two.i != &three.i).
// All three Examples share "num_instances", such that:
```

```
//  (&one.num_instances == &two.num_instances)
//  (&one.num_instances == &three.num_instances)
//  (&two.num_instances == &three.num_instances)
```

Static member variables are not considered to be defined inside the class, only declared, and thus have their definition outside the class definition; the programmer is allowed, but not required, to initialise static variables in their definition. When defining the member variables, the keyword `static` is omitted.

```
class Example {
    static int num_instances;               // Declaration.

  public:
    static std::string static_str;          // Declaration.

    // ...
};

int         Example::num_instances;         // Definition.  Zero-initialised.
std::string Example::static_str = "Hello."; // Definition.
```

Due to this, static variables can be incomplete types (apart from `void`), as long as they're later defined as a complete type.

```
struct ForwardDeclared;

class ExIncomplete {
    static ForwardDeclared fd;
    static ExIncomplete    i_contain_myself;
    static int             an_array[];
};

struct ForwardDeclared {};

ForwardDeclared ExIncomplete::fd;
ExIncomplete    ExIncomplete::i_contain_myself;
int             ExIncomplete::an_array[5];
```

Static member functions can be defined inside or outside the class definition, as with normal member functions. As with static member variables, the keyword `static` is omitted when defining static member functions outside the class definition.

```
// For Example above, either...
class Example {
    // ...

  public:
    static int static_func() { return num_instances; }

    // ...

    void set_str(const std::string& str) { static_str = str; }
};

// Or...

class Example { /* ... */ };

int  Example::static_func() { return num_instances; }
```

```
void Example::set_str(const std::string& str) { static_str = str; }
```

If a static member variable is declared `const` but not `volatile`, and is of an integral or enumeration type, it can be initialised at declaration, inside the class definition.

```
enum E { VAL = 5 };

struct ExConst {
    const static int ci = 5;              // Good.
    static const E ce = VAL;              // Good.
    const static double cd = 5;           // Error.
    static const volatile int cvi = 5;    // Error.

    const static double good_cd;
    static const volatile int good_cvi;
};

const double ExConst::good_cd = 5;        // Good.
const volatile int ExConst::good_cvi = 5; // Good.
```

Version ≥ C++11

As of C++11, static member variables of `LiteralType` types (types that can be constructed at compile time, according to `constexpr` rules) can also be declared as `constexpr`; if so, they must be initialised within the class definition.

```
struct ExConstexpr {
    constexpr static int ci = 5;                      // Good.
    static constexpr double cd = 5;                   // Good.
    constexpr static int carr[] = { 1, 1, 2 };        // Good.
    static constexpr ConstexprConstructibleClass c{}; // Good.
    constexpr static int bad_ci;                      // Error.
};

constexpr int ExConstexpr::bad_ci = 5;                // Still an error.
```

If a `const` or `constexpr` static member variable is *odr-used* (informally, if it has its address taken or is assigned to a reference), then it must still have a separate definition, outside the class definition. This definition is not allowed to contain an initialiser.

```
struct ExODR {
    static const int odr_used = 5;
};

// const int ExODR::odr_used;

const int* odr_user = & ExODR::odr_used; // Error; uncomment above line to resolve.
```

As static members aren't tied to a given instance, they can be accessed using the scope operator, `::`.

```
std::string str = Example::static_str;
```

They can also be accessed as if they were normal, non-static members. This is of historical significance, but is used less commonly than the scope operator to prevent confusion over whether a member is static or non-static.

```
Example ex;
std::string rts = ex.static_str;
```

Class members are able to access static members without qualifying their scope, as with non-static class members.

```
class ExTwo {
    static int num_instances;
    int my_num;

  public:
    ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

They cannot be `mutable`, nor would they need to be; as they aren't tied to any given instance, whether an instance is or isn't const doesn't affect static members.

```
struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

    ExDontNeedMutable() : immuta(-5), muta(-5) {}
};
int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // Error: Can't modify read-only object.
dnm.muta = 5;   // Good.  Mutable fields of const objects can be written.
dnm.i = 5;      // Good.  Static members can be written regardless of an instance's const-ness.
```

Static members respect access modifiers, just like non-static members.

```
class ExAccess {
    static int prv_int;

  protected:
    static int pro_int;

  public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // Error: int ExAccess::prv_int is private.
int x2 = ExAccess::pro_int; // Error: int ExAccess::pro_int is protected.
int x3 = ExAccess::pub_int; // Good.
```

As they aren't tied to a given instance, static member functions have no `this` pointer; due to this, they can't access non-static member variables unless passed an instance.

```
class ExInstanceRequired {
    int i;

  public:
    ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; }                         // Error.
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // Good.
};
```

Due to not having a `this` pointer, their addresses can't be stored in pointers-to-member-functions, and are instead stored in normal pointers-to-functions.

```
struct ExPointer {
           void nsfunc() {}
    static void  sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // Error.
    f_ptr p_sf = &ExPointer::sfunc; // Good.
```

Due to not having a `this` pointer, they also cannot be `const` or `volatile`, nor can they have ref-qualifiers. They also cannot be virtual.

```
struct ExCVQualifiersAndVirtual {
    static void   func()                {} // Good.
    static void  cfunc() const          {} // Error.
    static void  vfunc() volatile       {} // Error.
    static void cvfunc() const volatile {} // Error.
    static void  rfunc() &              {} // Error.
    static void rvfunc() &&             {} // Error.

    virtual static void vsfunc()        {} // Error.
    static virtual void svfunc()        {} // Error.
};
```

As they aren't tied to a given instance, static member variables are effectively treated as special global variables; they're created when the program starts, and destroyed when it exits, regardless of whether any instances of the class actually exist. Only a single copy of each static member variable exists (unless the variable is declared `thread_local` (C++11 or later), in which case there's one copy per thread).

Static member variables have the same linkage as the class, whether the class has external or internal linkage. Local classes and unnamed classes aren't allowed to have static members.

Section 34.13: Multiple Inheritance

Aside from single inheritance:

```
class A {};
class B : public A {};
```

You can also have multiple inheritance:

```
class A {};
class B {};
class C : public A, public B {};
```

C will now have inherit from A and B at the same time.

Note: this can lead to ambiguity if the same names are used in multiple inherited `class`s or `struct`s. Be careful!

Ambiguity in Multiple Inheritance

Multiple inheritance may be helpful in certain cases but, sometimes odd sort of problem encounters while using multiple inheritance.

For example: Two base classes have functions with same name which is not overridden in derived class and if you write code to access that function using object of derived class, compiler shows error because, it cannot determine which function to call. Here is a code for this type of ambiguity in multiple inheritance.

```
class base1
{
  public:
     void funtion( )
     { //code for base1 function }
};
class base2
{
    void function( )
     { // code for base2 function }
};

class derived : public base1, public base2
{

};

int main()
{
    derived obj;

  // Error because compiler can't figure out which function to call
  //either function( ) of base1 or base2 .
    obj.function( )
}
```

But, this problem can be solved using scope resolution function to specify which function to class either base1 or base2:

```
int main()
{
    obj.base1::function( );  // Function of class base1 is called.
    obj.base2::function( );  // Function of class base2 is called.
}
```

Section 34.14: Non-static member functions

A class can have non-static member functions, which operate on individual instances of the class.

```
class CL {
  public:
    void member_function() {}
};
```

These functions are called on an instance of the class, like so:

```
CL instance;
instance.member_function();
```

They can be defined either inside or outside the class definition; if defined outside, they are specified as being in the class' scope.

```
struct ST {
    void  defined_inside() {}
    void defined_outside();
};
void ST::defined_outside() {}
```

They can be CV-qualified and/or ref-qualified, affecting how they see the instance they're called upon; the function will see the instance as having the specified cv-qualifier(s), if any. Which version is called will be based on the instance's cv-qualifiers. If there is no version with the same cv-qualifiers as the instance, then a more-cv-qualified version will be called if available.

```
struct CVQualifiers {
    void func()                   {} // 1: Instance is non-cv-qualified.
    void func() const             {} // 2: Instance is const.

    void cv_only() const volatile {}
};

CVQualifiers       non_cv_instance;
const CVQualifiers      c_instance;

non_cv_instance.func(); // Calls #1.
c_instance.func();      // Calls #2.

non_cv_instance.cv_only(); // Calls const volatile version.
c_instance.cv_only();      // Calls const volatile version.
```

Version ≥ C++11

Member function ref-qualifiers indicate whether or not the function is intended to be called on rvalue instances, and use the same syntax as function cv-qualifiers.

```
struct RefQualifiers {
    void func() &  {} // 1: Called on normal instances.
    void func() && {} // 2: Called on rvalue (temporary) instances.
};

RefQualifiers rf;
rf.func();              // Calls #1.
RefQualifiers{}.func(); // Calls #2.
```

CV-qualifiers and ref-qualifiers can also be combined if necessary.

```
struct BothCVAndRef {
    void func() const& {} // Called on normal instances.  Sees instance as const.
    void func() &&     {} // Called on temporary instances.
};
```

They can also be virtual; this is fundamental to polymorphism, and allows a child class(es) to provide the same interface as the parent class, while supplying their own functionality.

```
struct Base {
    virtual void func() {}
};
struct Derived {
    virtual void func() {}
};

Base* bp = new Base;
Base* dp = new Derived;
bp.func(); // Calls Base::func().
dp.func(); // Calls Derived::func().
```

For more information, see here.

Chapter 35: Function Overloading

See also separate topic on Overload Resolution

Section 35.1: What is Function Overloading?

Function overloading is having multiple functions declared in the same scope with the exact same name exist in the same place (known as *scope*) differing only in their *signature*, meaning the arguments they accept.

Suppose you are writing a series of functions for generalized printing capabilities, beginning with `std::string`:

```
void print(const std::string &str)
{
    std::cout << "This is a string: " << str << std::endl;
}
```

This works fine, but suppose you want a function that also accepts an `int` and prints that too. You could write:

```
void print_int(int num)
{
    std::cout << "This is an int:  " << num << std::endl;
}
```

But because the two functions accept different parameters, you can simply write:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Now you have 2 functions, both named `print`, but with different signatures. One accepts `std::string`, the other one an `int`. Now you can call them without worrying about different names:

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337);           //prints "This is an int: 1337"
```

Instead of:

```
print("Hello world!");
print_int(1337);
```

When you have overloaded functions, the compiler infers which of the functions to call from the parameters you provide it. Care must be taken when writing function overloads. For example, with implicit type conversions:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
void print(double num)
{
    std::cout << "This is a double: " << num << std::endl;
}
```

Now it's not immediately clear which overload of `print` is called when you write:

```
print(5);
```

And you might need to give your compiler some clues, like:

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

Some care also needs to be taken when writing overloads that accept optional parameters:

```
// WRONG CODE
void print(int num1, int num2 = 0)    //num2 defaults to 0 if not included
{
    std::cout << "These are ints: << num1 << " and " << num2 << std::endl;
}
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Because there's no way for the compiler to tell if a call like `print(17)` is meant for the first or second function because of the optional second parameter, this will fail to compile.

Section 35.2: Return Type in Function Overloading

Note that you cannot overload a function based on its return type. For example:

```
// WRONG CODE
std::string getValue()
{
  return "hello";
}

int getValue()
{
  return 0;
}

int x = getValue();
```

This will cause a compilation error as the compiler will not be able to work out which version of `getValue` to call, even though the return type is assigned to an `int`.

Section 35.3: Member Function cv-qualifier Overloading

Functions within a class can be overloaded for when they are accessed through a cv-qualified reference to that class; this is most commonly used to overload for `const`, but can be used to overload for `volatile` and `const volatile`, too. This is because all non-static member functions take `this` as a hidden parameter, which the cv-qualifiers are applied to. This is most commonly used to overload for `const`, but can also be used for `volatile` and `const volatile`.

This is necessary because a member function can only be called if it is at least as cv-qualified as the instance it's called on. While a non-`const` instance can call both `const` and non-`const` members, a `const` instance can only call `const` members. This allows a function to have different behaviour depending on the calling instance's cv-qualifiers, and allows the programmer to disallow functions for an undesired cv-qualifier(s) by not providing a version with that qualifier(s).

A class with some basic `print` method could be `const` overloaded like so:

```
#include <iostream>

class Integer
{
    public:
        Integer(int i_): i{i_}{}

        void print()
        {
            std::cout << "int: " << i << std::endl;
        }

        void print() const
        {
            std::cout << "const int: " << i << std::endl;
        }

    protected:
        int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // prints "int: 5"
    ic.print(); // prints "const int: 5"
}
```

This is a key tenet of `const` correctness: By marking member functions as `const`, they are allowed to be called on `const` instances, which in turn allows functions to take instances as `const` pointers/references if they don't need to modify them. This allows code to specify whether it modifies state by taking unmodified parameters as `const` and modified parameters without cv-qualifiers, making code both safer and more readable.

```
class ConstCorrect
{
  public:
    void good_func() const
    {
        std::cout << "I care not whether the instance is const." << std::endl;
    }

    void bad_func()
    {
        std::cout << "I can only be called on non-const, non-volatile instances." << std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "I can take either a const or a non-const ConstCorrect." << std::endl;
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Error.  Can only be called from non-const instance.
}

void const_incorrect_func(ConstCorrect& cc)
{
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Good.  Can only be called from non-const instance.
}
```

A common usage of this is declaring accessors as `const`, and mutators as non-`const`.

No class members can be modified within a `const` member function. If there is some member that you really need to modify, such as locking a `std::mutex`, you can declare it as `mutable`:

```
class Integer
{
    public:
        Integer(int i_): i{i_}{}

        int get() const
        {
            std::lock_guard<std::mutex> lock{mut};
            return i;
        }

        void set(int i_)
        {
            std::lock_guard<std::mutex> lock{mut};
            i = i_;
        }

    protected:
        int i;
        mutable std::mutex mut;
};
```

Chapter 36: Operator Overloading

In C++, it is possible to define operators such as + and -> for user-defined types. For example, the **<string>** header defines a + operator to concatenate strings. This is done by defining an *operator function* using the operator keyword.

Section 36.1: Arithmetic operators

You can overload all basic arithmetic operators:

- + and +=
- - and -=
- * and *=
- / and /=
- & and &=
- | and |=
- ^ and ^=
- >> and >>=
- << and <<=

Overloading for all operators is the same. *Scroll down for explanation*

Overloading outside of class/struct:

```
//operator+ should be implemented in terms of operator+=
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //Perform addition
    return lhs;
}
```

Overloading inside of class/struct:

```
//operator+ should be implemented in terms of operator+=
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //Perform addition
    return *this;
}
```

Note: operator+ should return by non-const value, as returning a reference wouldn't make sense (it returns a *new* object) nor would returning a const value (you should generally not return by const). The first argument is passed by value, why? Because

1. You can't modify the original object (`Object foobar = foo + bar;` shouldn't modify `foo` after all, it wouldn't make sense)
2. You can't make it `const`, because you will have to be able to modify the object (because `operator+` is implemented in terms of `operator+=`, which modifies the object)

Passing by `const&` would be an option, but then you will have to make a temporary copy of the passed object. By passing by value, the compiler does it for you.

`operator+=` returns a reference to the itself, because it is then possible to chain them (don't use the same variable though, that would be undefined behavior due to sequence points).

The first argument is a reference (we want to modify it), but not `const`, because then you wouldn't be able to modify it. The second argument should not be modified, and so for performance reason is passed by `const&` (passing by const reference is faster than by value).

Section 36.2: Array subscript operator

You can even overload the array subscript operator `[]`.

You should **always** (99.98% of the time) implement 2 versions, a `const` and a not-`const` version, because if the object is `const`, it should not be able to modify the object returned by `[]`.

The arguments are passed by `const&` instead of by value because passing by reference is faster than by value, and `const` so that the operator doesn't change the index accidentally.

The operators return by reference, because by design you can modify the object `[]` return, i.e:

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
          //wouldn't be possible if not returned by reference
```

You can **only** overload inside a `class`/`struct`:

```
//I is the index type, normally an int
T& operator[](const I& index)
{
    //Do something
    //return something
}

//I is the index type, normally an int
const T& operator[](const I& index) const
{
    //Do something
    //return something
}
```

Multiple subscript operators, `[][]...`, can be achieved via proxy objects. The following example of a simple row-major matrix class demonstrates this:

```
template<class T>
```

```
class matrix {
    // class enabling [][] overload to access matrix elements
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
            return vec[row_index*cols + _col_index];
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
    private:
        C& vec;
        std::size_t row_index; // row index to access
        std::size_t cols; // number of columns in matrix
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // call operator[] followed by another [] call to access matrix elements
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};
```

Section 36.3: Conversion operators

You can overload type operators, so that your type can be implicitly converted into the specified type.

The conversion operator **must** be defined in a `class`/`struct`:

```
operator T() const { /* return something */ }
```

Note: the operator is `const` to allow `const` objects to be converted.

Example:

```
struct Text
{
    std::string text;

    // Now Text can be implicitly converted into a const char*
    /*explicit*/ operator const char*() const { return text.data(); }
    // ^^^^^^^
    // to disable implicit conversion
};

Text t;
t.text = "Hello world!";

//Ok
const char* copyoftext = t;
```

Section 36.4: Complex Numbers Revisited

The code below implements a very simple complex number type for which the underlying field is automatically promoted, following the language's type promotion rules, under application of the four basic operators (+, -, *, and /) with a member of a different field (be it another `complex<T>` or some scalar type).

This is intended to be a holistic example covering operator overloading alongside basic use of templates.

```
#include <type_traits>

namespace not_std{

using std::decay_t;

//-----------------------------------------------------------------
// complex< value_t >
//-----------------------------------------------------------------

template<typename value_t>
struct complex
{
    value_t x;
    value_t y;

    complex &operator += (const value_t &x)
    {
        this->x += x;
        return *this;
    }
    complex &operator += (const complex &other)
    {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    complex &operator -= (const value_t &x)
    {
        this->x -= x;
        return *this;
    }
    complex &operator -= (const complex &other)
    {
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }

    complex &operator *= (const value_t &s)
    {
        this->x *= s;
        this->y *= s;
        return *this;
    }
    complex &operator *= (const complex &other)
    {
        (*this) = (*this) * other;
        return *this;
    }

    complex &operator /= (const value_t &s)
    {
        this->x /= s;
        this->y /= s;
        return *this;
    }
    complex &operator /= (const complex &other)
    {
        (*this) = (*this) / other;
        return *this;
    }

    complex(const value_t &x, const value_t &y)
    : x{x}
    , y{y}
    {}

    template<typename other_value_t>
    explicit complex(const complex<other_value_t> &other)
    : x{static_cast<const value_t &>(other.x)}
    , y{static_cast<const value_t &>(other.y)}
    {}

    complex &operator = (const complex &) = default;
    complex &operator = (complex &&) = default;
    complex(const complex &) = default;
    complex(complex &&) = default;
    complex() = default;
};

// Absolute value squared
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//-----------------------------------------------------------------
// operator - (negation)
//-----------------------------------------------------------------

template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }

//-----------------------------------------------------------------
// operator +
//-----------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }
```

```
template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//----------------------------------------------------------------
// operator -
//----------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return{a.x - b.x, a.y - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return{a - b.x, - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//----------------------------------------------------------------
// operator *
//----------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
        };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
{ return {a.x * b, a.y * b}; }

//----------------------------------------------------------------
// operator /
//----------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>
{
    const auto r = absqr(b);
    return {
        ( a.x*b.x + a.y*b.y) / r,
        (-a.x*b.y + a.y*b.x) / r
        };
}

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
         b.x * s,
        -b.y * s
        };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

}// namespace not_std


int main(int argc, char **argv)
{
    using namespace not_std;

    complex<float> fz{4.0f, 1.0f};

    // makes a complex<double>
    auto dz = fz * 1.0;

    // still a complex<double>
    auto idz = 1.0f/dz;

    // also a complex<double>
    auto one = dz * idz;

    // a complex<double> again
    auto one_again = fz * idz;

    // Operator tests, just to make sure everything compiles.

    complex<float> a{1.0f, -2.0f};
    complex<double> b{3.0, -4.0};

    // All of these are complex<double>
    auto c0 = a + b;
    auto c1 = a - b;
    auto c2 = a * b;
    auto c3 = a / b;

    // All of these are complex<float>
    auto d0 = a + 1;
    auto d1 = 1 + a;
    auto d2 = a - 1;
    auto d3 = 1 - a;
    auto d4 = a * 1;
    auto d5 = 1 * a;
    auto d6 = a / 1;
    auto d7 = 1 / a;

    // All of these are complex<double>
    auto e0 = b + 1;
    auto e1 = 1 + b;
    auto e2 = b - 1;
    auto e3 = 1 - b;
    auto e4 = b * 1;
    auto e5 = 1 * b;
    auto e6 = b / 1;
    auto e7 = 1 / b;

    return 0;
}
```

Section 36.5: Named operators

You can extend C++ with named operators that are "quoted" by standard C++ operators.

First we start with a dozen-line library:

```
namespace named_operator {
  template<class D>struct make_operator{constexpr make_operator(){}};

  template<class T, char, class O> struct half_apply { T&& lhs; };

  template<class Lhs, class Op>
  half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
    return {std::forward<Lhs>(lhs)};
  }

  template<class Lhs, class Op, class Rhs>
  auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
  -> decltype( named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) ) )
  {
    return named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) );
  }
}
```

this doesn't do anything yet.

First, appending vectors

```
namespace my_ns {
  struct append_t : named_operator::make_operator<append_t> {};
  constexpr append_t append{};

  template<class T, class A0, class A1>
  std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const& rhs
) {
      lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
      return std::move(lhs);
  }
}
using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};
```

```
auto c = a *append* b;
```

The core here is that we define an `append` object of type `append_t:named_operator::make_operator<append_t>`.

We then overload named_invoke(lhs, append_t, rhs) for the types we want on the right and left.

The library overloads `lhs*append_t`, returning a temporary `half_apply` object. It also overloads `half_apply*rhs` to call `named_invoke( lhs, append_t, rhs )`.

We simply have to create the proper `append_t` token and do an ADL-friendly `named_invoke` of the proper signature, and everything hooks up and works.

For a more complex example, suppose you want to have element-wise multiplication of elements of a std::array:

```
template<class=void, std::size_t...Is>
auto indexer( std::index_sequence<Is...> ) {
  return [](auto&& f) {
    return f( std::integral_constant<std::size_t, Is>{}... );
  };
}
template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
  struct e_times_t : named_operator::make_operator<e_times_t> {};
  constexpr e_times_t e_times{};

  template<class L, class R, std::size_t N,
    class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
  >
  std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N> const&
rhs ) {
    using result_type = std::array<Out, N>;
    auto index_over_N = indexer<N>();
    return index_over_N([&](auto...is)->result_type {
      return {{
        (lhs[is] * rhs[is])...
      }};
    });
  }
}
```

live example.

This element-wise array code can be extended to work on tuples or pairs or C-style arrays, or even variable length containers if you decide what to do if the lengths don't match.

You could also an element-wise operator type and get `lhs *element_wise<'+'>* rhs`.

Writing a `*dot*` and `*cross*` product operators are also obvious uses.

The use of `*` can be extended to support other delimiters, like `+`. The delimeter precidence determines the precidence of the named operator, which may be important when translating physics equations over to C++ with minimal use of extra `()`s.

With a slight change in the library above, we can support `->*then*` operators and extend `std::function` prior to the standard being updated, or write monadic `->*bind*`. It could also have a stateful named operator, where we carefully pass the `Op` down to the final invoke function, permitting:

```
named_operator<'*'> append = [](auto lhs, auto&& rhs) {
  using std::begin; using std::end;
  lhs.insert( end(lhs), begin(rhs), end(rhs) );
  return std::move(lhs);
};
```

generating a named container-appending operator in C++17.

Section 36.6: Unary operators

You can overload the 2 unary operators:

- `++foo` and `foo++`
- `--foo` and `foo--`

Overloading is the same for both types (`++` and `--`). *Scroll down for explanation*

Overloading outside of `class`/`struct`:

```
//Prefix operator ++foo
T& operator++(T& lhs)
{
    //Perform addition
    return lhs;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}
```

Overloading inside of `class`/`struct`:

```
//Prefix operator ++foo
T& operator++()
{
    //Perform addition
    return *this;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}
```

Note: The prefix operator returns a reference to itself, so that you can continue operations on it. The first argument is a reference, as the prefix operator changes the object, that's also the reason why it isn't `const` (you wouldn't be able to modify it otherwise).

The postfix operator returns by value a temporary (the previous value), and so it cannot be a reference, as it would be a reference to a temporary, which would be garbage value at the end of the function, because the temporary variable goes out of scope). It also cannot be `const`, because you should be able to modify it directly.

The first argument is a non-`const` reference to the "calling" object, because if it were `const`, you wouldn't be able to modify it, and if it weren't a reference, you wouldn't change the original value.

It is because of the copying needed in postfix operator overloads that it's better to make it a habit to use prefix ++ instead of postfix ++ in `for` loops. From the `for` loop perspective, they're usually functionally equivalent, but there might be a slight performance advantage to using prefix ++, especially with "fat" classes with a lot of members to copy. Example of using prefix ++ in a for loop:

```
for (list<string>::const_iterator it = tokens.begin();
     it != tokens.end();
     ++it) { // Don't use it++
    ...
}
```

Section 36.7: Comparison operators

You can overload all comparison operators:

- `==` and `!=`
- `>` and `<`
- `>=` and `<=`

The recommended way to overload all those operators is by implementing only 2 operators (`==` and `<`) and then using those to define the rest. *Scroll down for explanation*

Overloading outside of `class`/`struct`:

```
//Only implement those 2
bool operator==(const T& lhs, const T& rhs) { /* Compare */ }
bool operator<(const T& lhs, const T& rhs) { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

Overloading inside of `class`/`struct`:

```
//Note that the functions are const, because if they are not const, you wouldn't be able
//to call them if the object is const

//Only implement those 2
bool operator==(const T& rhs) const { /* Compare */ }
bool operator<(const T& rhs) const { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& rhs) const { return !(*this == rhs); }
bool operator>(const T& rhs) const { return rhs < *this; }
bool operator<=(const T& rhs) const { return !(*this > rhs); }
bool operator>=(const T& rhs) const { return !(*this < rhs); }
```

The operators obviously return a `bool`, indicating `true` or `false` for the corresponding operation.

All of the operators take their arguments by `const&`, because the only thing that does operators do is compare, so they shouldn't modify the objects. Passing by `&` (reference) is faster than by value, and to make sure that the operators don't modify it, it is a `const`-reference.

Note that the operators inside the `class`/`struct` are defined as `const`, the reason for this is that without the functions being `const`, comparing `const` objects would not be possible, as the compiler doesn't know that the operators don't modify anything.

Section 36.8: Assignment operator

The assignment operator is one of the most important operators because it allows you to change the status of a variable.

If you do not overload the assignment operator for your `class`/`struct`, it is automatically generated by the compiler: the automatically-generated assignment operator performs a "memberwise assignment", ie by invoking assignment operators on all members, so that one object is copied to the other, a member at time. The assignment operator should be overloaded when the simple memberwise assignment is not suitable for your `class`/`struct`, for example if you need to perform a **deep copy** of an object.

Overloading the assignment operator `=` is easy, but you should follow some simple steps.

1. **Test for self-assignment.** This check is important for two reasons:
    - a self-assignment is a needless copy, so it does not make sense to perform it;
    - the next step will not work in the case of a self-assignment.
2. **Clean the old data.** The old data must be replaced with new ones. Now, you can understand the second reason of the previous step: if the content of the object was destroyed, a self-assignment will fail to perform the copy.
3. **Copy all members.** If you overload the assignment operator for your `class` or your `struct`, it is not automatically generated by the compiler, so you will need to take charge of copying all members from the other object.
4. **Return `*this`.** The operator returns by itself by reference, because it allows chaining (i.e. `int b = (a = 6) + 4; //b == 10`).

```
//T is some type
T& operator=(const T& other)
{
    //Do something (like copying values)
    return *this;
}
```

Note: `other` is passed by `const&`, because the object being assigned should not be changed, and passing by reference is faster than by value, and to make sure than `operator=` doesn't modify it accidentally, it is `const`.

The assignment operator can **only** to be overloaded in the `class`/`struct`, because the left value of `=` is **always** the `class`/`struct` itself. Defining it as a free function doesn't have this guarantee, and is disallowed because of that.

When you declare it in the `class`/`struct`, the left value is implicitly the `class`/`struct` itself, so there is no problem with that.

Section 36.9: Function call operator

You can overload the function call operator `()`:

Overloading must be done inside of a `class`/`struct`:

```
//R -> Return type
//Types -> any different type
R operator()(Type name, Type2 name2, ...)
{
    //Do something
    //return something
}

//Use it like this (R is return type, a and b are variables)
R foo = object(a, b, ...);
```

For example:

```
struct Sum
{
    int operator()(int a, int b)
    {
        return a + b;
    }
};

//Create instance of struct
Sum sum;
int result = sum(1, 1); //result == 2
```

Section 36.10: Bitwise NOT operator

Overloading the bitwise NOT (~) is fairly simple. *Scroll down for explanation*

Overloading outside of `class`/`struct`:

```
T operator~(T lhs)
{
    //Do operation
    return lhs;
}
```

Overloading inside of `class`/`struct`:

```
T operator~()
{
    T t(*this);
    //Do operation
    return t;
}
```

Note: `operator~` returns by value, because it has to return a new value (the modified value), and not a reference to the value (it would be a reference to the temporary object, which would have garbage value in it as soon as the operator is done). Not `const` either because the calling code should be able to modify it afterwards (i.e. `int a = ~a + 1;` should be possible).

Inside the class/struct you have to make a temporary object, because you can't modify this, as it would modify the original object, which shouldn't be the case.

Section 36.11: Bit shift operators for I/O

The operators << and >> are commonly used as "write" and "read" operators:

- std::ostream overloads << to write variables to the underlying stream (example: std::cout)
- std::istream overloads >> to read from the underlying stream to a variable (example: std::cin)

The way they do this is similar if you wanted to overload them "normally" outside of the class/struct, except that specifying the arguments are not of the same type:

- Return type is the stream you want to overload from (for example, std::ostream) passed by reference, to allow chaining (Chaining: std::cout << a << b;). Example: std::ostream&
- lhs would be the same as the return type
- rhs is the type you want to allow overloading from (i.e. T), passed by const& instead of value for performance reason (rhs shouldn't be changed anyway). Example: const Vector&.

Example:

```
//Overload std::ostream operator<< to allow output from Vector's
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
    lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << '\n';
    return lhs;
}

Vector v = { 1, 2, 3};

//Now you can do
std::cout << v;
```

Chapter 37: Function Template Overloading

Section 37.1: What is a valid function template overloading?

A function template can be overloaded under the rules for non-template function overloading (same name, but different parameter types) and in addition to that, the overloading is valid if

- The return type is different, or
- The template parameter list is different, except for the naming of parameters and the presence of default arguments (they are not part of the signature)

For a normal function, comparing two parameter types is is easy for the compiler, since it has all informat. But a type within a template may not be determined yet. Therefore, the rule for when two parameter types are equal is approximative here and says that the non-depependend types and values need to match and the spelling of dependent types and expressions needs to be the same (more precisely, they need to conform to the so-called ODR-rules), except that template parameters may be renamed. However, if under such different spellings, two values within the types are deemed different, but will always instantiate to the same values, the overloading is invalid, but no diagnostic is required from the compiler.

```
template<typename T>
void f(T*) { }

template<typename T>
void f(T) { }
```

This is a valid overload, as "T" and "T*" are different spellings. But the following is invalid, with no diagnostic required

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }

template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

Chapter 38: Virtual Member Functions

Section 38.1: Final virtual functions

C++11 introduced `final` specifier which forbids method overriding if appeared in method signature:

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo\n";
    }
};

class Derived1 : public Base {
public:
    // Overriding Base::foo
    void foo() final {
        std::cout << "Derived1::Foo\n";
    }
};

class Derived2 : public Derived1 {
public:
    // Compilation error: cannot override final method
    virtual void foo() {
        std::cout << "Derived2::Foo\n";
    }
};
```

The specifier `final` can only be used with `virtual' member function and can't be applied to non-virtual member functions

Like `final`, there is also an specifier caller 'override' which prevent overriding of `virtual` functions in the derived class.

The specifiers `override` and `final` may be combined together to have desired effect:

```
class Derived1 : public Base {
public:
    void foo() final override {
        std::cout << "Derived1::Foo\n";
    }
};
```

Section 38.2: Using override with virtual in C++11 and later

The specifier `override` has a special meaning in C++11 onwards, if appended at the end of function signature. This signifies that a function is

- Overriding the function present in base class &
- The Base class function is `virtual`

There is no `run time` significance of this specifier as is mainly meant as an indication for compilers

The example below will demonstrate the change in behaviour with our without using override.

Without `override`:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Y::f() will not override X::f() because it has a different signature,
    // but the compiler will accept the code (and silently ignore Y::f()).
    virtual void f(int a) { std::cout << a << "\n"; }
};
```

With `override`:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // The compiler will alert you to the fact that Y::f() does not
    // actually override anything.
    virtual void f(int a) override { std::cout << a << "\n"; }
};
```

Note that `override` is not a keyword, but a special identifier which only may appear in function signatures. In all other contexts `override` still may be used as an identifier:

```
void foo() {
    int override = 1; // OK.
    int virtual = 2;  // Compilation error: keywords can't be used as identifiers.
}
```

Section 38.3: Virtual vs non-virtual member functions

With virtual member functions:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Specifying virtual again here is optional
    // because it can be inferred from X::f().
    virtual void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "Y::f()"
}
```

Without virtual member functions:

```
#include <iostream>

struct X {
   void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
   void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "X::f()"
}
```

Section 38.4: Behaviour of virtual functions in constructors and destructors

The behaviour of virtual functions in constructors and destructors is often confusing when first encountered.

```
#include <iostream>
using namespace std;

class base {
public:
    base() { f("base constructor"); }
    ~base() { f("base destructor"); }

    virtual const char* v() { return "base::v()"; }

    void f(const char* caller) {
        cout << "When called from " << caller << ", "  << v() << " gets called.\n";
    }
};

class derived : public base {
public:
    derived() { f("derived constructor"); }
    ~derived() { f("derived destructor"); }

    const char* v() override { return "derived::v()"; }

};

int main() {
     derived d;
}
```

Output:

> When called from base constructor, base::v() gets called.
> When called from derived constructor, derived::v() gets called.
> When called from derived destructor, derived::v() gets called.
> When called from base destructor, base::v() gets called.

The reasoning behind this is that the derived class may define additional members which are not yet initialized (in the constructor case) or already destroyed (in the destructor case), and calling its member functions would be unsafe. Therefore during construction and destruction of C++ objects, the *dynamic* type of `*this` is considered to be the constructor's or destructor's class and not a more-derived class.

Example:

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

using namespace std;
class base {
public:
    base()
    {
        std::cout << "foo is " << foo() << std::endl;
    }
    virtual int foo() { return 42; }
};

class derived : public base {
    unique_ptr<int> ptr_;
public:
    derived(int i) : ptr_(new int(i*i)) { }
    // The following cannot be called before derived::derived due to how C++ behaves,
    // if it was possible... Kaboom!
    int foo() override   { return *ptr_; }
};

int main() {
    derived d(4);
}
```

Section 38.5: Pure virtual functions

We can also specify that a `virtual` function is *pure virtual* (abstract), by appending `= 0` to the declaration. Classes with one or more pure virtual functions are considered to be abstract, and cannot be instantiated; only derived classes which define, or inherit definitions for, all pure virtual functions can be instantiated.

```
struct Abstract {
    virtual void f() = 0;
};

struct Concrete {
    void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.
```

Even if a function is specified as pure virtual, it can be given a default implementation. Despite this, the function will still be considered abstract, and derived classes will have to define it before they can be instantiated. In this case, the derived class' version of the function is even allowed to call the base class' version.

```
struct DefaultAbstract {
    virtual void f() = 0;
};
void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};
```

There are a couple of reasons why we might want to do this:

- If we want to create a class that can't itself be instantiated, but doesn't prevent its derived classes from being instantiated, we can declare the destructor as pure virtual. Being the destructor, it must be defined anyways, if we want to be able to deallocate the instance. And as the destructor is most likely already virtual to prevent memory leaks during polymorphic use, we won't incur an unnecessary performance hit from declaring another function `virtual`. This can be useful when making interfaces.

  ```
  struct Interface {
      virtual ~Interface() = 0;
  };
  Interface::~Interface() = default;

  struct Implementation : Interface {};
  // ~Implementation() is automatically defined by the compiler if not explicitly
  //  specified, meeting the "must be defined before instantiation" requirement.
  ```

- If most or all implementations of the pure virtual function will contain duplicate code, that code can instead be moved to the base class version, making the code easier to maintain.

  ```
  class SharedBase {
      State my_state;
      std::unique_ptr<Helper> my_helper;
      // ...

    public:
      virtual void config(const Context& cont) = 0;
      // ...
  };
  /* virtual */ void SharedBase::config(const Context& cont) {
      my_helper = new Helper(my_state, cont.relevant_field);
      do_this();
      and_that();
  }

  class OneImplementation : public SharedBase {
      int i;
      // ...

    public:
      void config(const Context& cont) override;
      // ...
  };
  void OneImplementation::config(const Context& cont) /* override */ {
      my_state = { cont.some_field, cont.another_field, i };
      SharedBase::config(cont);
      my_unique_setup();
  };
  ```

```
  // And so on, for other classes derived from SharedBase.
```

Chapter 39: Inline functions

A function defined with the `inline` specifier is an inline function. An inline function can be multiply defined without violating the One Definition Rule, and can therefore be defined in a header with external linkage. Declaring a function inline hints to the compiler that the function should be inlined during code generation, but does not provide a guarantee.

Section 39.1: Non-member inline function definition

```
inline int add(int x, int y)
{
    return x + y;
}
```

Section 39.2: Member inline functions

```
// header (.hpp)
struct A
{
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
{
}
```

Section 39.3: What is function inlining?

```
inline int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

In the above code, when `add` is inlined, the resulting code would become something like this

```
int main()
{
    int a = 1, b = 2;
    int c = a + b;
}
```

The inline function is nowhere to be seen, its body gets *inlined* into the caller's body. Had `add` not been inlined, a function would be called. The overhead of calling a function -- such as creating a new stack frame, copying arguments, making local variables, jump (losing locality of reference and there by cache misses), etc. -- has to be incurred.

Section 39.4: Non-member inline function declaration

```
inline int add(int x, int y);
```

Chapter 40: Special Member Functions

Section 40.1: Default Constructor

A *default constructor* is a type of constructor that requires no parameters when called. It is named after the type it constructs and is a member function of it (as all constructors are).

```
class C{
    int i;
public:
    // the default constructor definition
    C()
    : i(0){ // member initializer list -- initialize i to 0
        // constructor function body -- can do more complex things here
    }
};

C c1; // calls default constructor of C to create object c1
C c2 = C(); // calls default constructor explicitly
C c3(); // ERROR: this intuitive version is not possible due to "most vexing parse"
C c4{}; // but in C++11 {} CAN be used in a similar way

C c5[2]; // calls default constructor for both array elements
C* c6 = new C[2]; // calls default constructor for both array elements
```

Another way to satisfy the "no parameters" requirement is for the developer to provide default values for all parameters:

```
class D{
    int i;
    int j;
public:
    // also a default constructor (can be called with no parameters)
    D( int i = 0, int j = 42 )
    : i(i), j(j){
    }
};


D d; // calls constructor of D with the provided default values for the parameters
```

Under some circumstances (i.e., the developer provides no constructors and there are no other disqualifying conditions), the compiler implicitly provides an empty default constructor:

```
class C{
    std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

Having some other type of constructor is one of the disqualifying conditions mentioned earlier:

```
class C{
    int i;
public:
    C( int i ) : i(i){}
};
```

```
C c1; // Compile ERROR: C has no (implicitly defined) default constructor
```

Version < c++11

To prevent implicit default constructor creation, a common technique is to declare it as `private` (with no definition). The intention is to cause a compile error when someone tries to use the constructor (this either results in an *Access to private* error or a linker error, depending on the compiler).

To be sure a default constructor (functionally similar to the implicit one) is defined, a developer could write an empty one explicitly.

Version ≥ c++11

In C++11, a developer can also use the `delete` keyword to prevent the compiler from providing a default constructor.

```
class C{
    int i;
public:
    // default constructor is explicitly deleted
    C() = delete;
};

C c1; // Compile ERROR: C has its default constructor deleted
```

Furthermore, a developer may also be explicit about wanting the compiler to provide a default constructor.

```
class C{
    int i;
public:
    // does have automatically generated default constructor (same as implicit one)
    C() = default;

    C( int i ) : i(i){}
};

C c1; // default constructed
C c2( 1 ); // constructed with the int taking constructor
```

Version ≥ c++14

You can determine whether a type has a default constructor (or is a primitive type) using `std::is_default_constructible` from **`<type_traits>`**:

```
class C1{ };
class C2{ public: C2(){} };
class C3{ public: C3(int){} };

using std::cout; using std::boolalpha; using std::endl;
using std::is_default_constructible;
cout << boolalpha << is_default_constructible<int>() << endl; // prints true
cout << boolalpha << is_default_constructible<C1>() << endl; // prints true
cout << boolalpha << is_default_constructible<C2>() << endl; // prints true
cout << boolalpha << is_default_constructible<C3>() << endl; // prints false
```

Version = c++11

In C++11, it is still possible to use the non-functor version of `std::is_default_constructible`:

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

Section 40.2: Destructor

A *destructor* is a function without arguments that is called when a user-defined object is about to be destroyed. It is named after the type it destructs with a ~ prefix.

```
class C{
    int* is;
    string s;
public:
    C()
    : is( new int[10] ){
    }

    ~C(){  // destructor definition
        delete[] is;
    }
};

class C_child : public C{
    string s_ch;
public:
    C_child(){}
    ~C_child(){} // child destructor
};

void f(){
    C c1; // calls default constructor
    C c2[2]; // calls default constructor for both elements
    C* c3 = new C[2]; // calls default constructor for both array elements

    C_child c_ch;  // when destructed calls destructor of s_ch and of C base (and in turn s)

    delete[] c3; // calls destructors on c3[0] and c3[1]
} // automatic variables are destroyed here -- i.e. c1, c2 and c_ch
```

Under most circumstances (i.e., a user provides no destructor, and there are no other disqualifying conditions), the compiler provides a default destructor implicitly:

```
class C{
    int i;
    string s;
};

void f(){
    C* c1 = new C;
    delete c1; // C has a destructor
}
```

```
class C{
    int m;
private:
    ~C(){} // not public destructor!
};

class C_container{
    C c;
};

void f(){
    C_container* c_cont = new C_container;
    delete c_cont; // Compile ERROR: C has no accessible destructor
}
```

Version > c++11

In C++11, a developer can override this behavior by preventing the compiler from providing a default destructor.

```
class C{
    int m;
public:
    ~C() = delete; // does NOT have implicit destructor
};

void f{
    C c1;
} // Compile ERROR: C has no destructor
```

Furthermore, a developer may also be explicit about wanting the compiler to provide a default destructor.

```
class C{
    int m;
public:
    ~C() = default; // saying explicitly it does have implicit/empty destructor
};

void f(){
    C c1;
} // C has a destructor -- c1 properly destroyed
```

Version > c++11

You can determine whether a type has a destructor (or is a primitive type) using `std::is_destructible` from **`<type_traits>`**:

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // prints true
cout << boolalpha << is_destructible<C1>() << endl; // prints true
cout << boolalpha << is_destructible<C2>() << endl; // prints false
cout << boolalpha << is_destructible<C3>() << endl; // prints false
```

Section 40.3: Copy and swap

If you're writing a class that manages resources, you need to implement all the special member functions (see Rule of Three/Five/Zero). The most direct approach to writing the copy constructor and assignment operator would be:

```
person(const person &other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        delete [] name;
        name = new char[std::strlen(other.name) + 1];
        std::strcpy(name, other.name);
        age = other.age;
    }

    return *this;
}
```

But this approach has some problems. It fails the strong exception guarantee - if `new[]` throws, we've already cleared the resources owned by `this` and cannot recover. We're duplicating a lot of the logic of copy construction in copy assignment. And we have to remember the self-assignment check, which usually just adds overhead to the copy operation, but is still critical.

To satisfy the strong exception guarantee and avoid code duplication (double so with the subsequent move assignment operator), we can use the copy-and-swap idiom:

```
class person {
    char* name;
    int age;
public:
    /* all the other functions ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // enable ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};
```

Why does this work? Consider what happens when we have

```
person p1 = ...;
person p2 = ...;
p1 = p2;
```

First, we copy-construct `rhs` from `p2` (which we didn't have to duplicate here). If that operation throws, we don't do anything in `operator=` and `p1` remains untouched. Next, we swap the members between `*this` and `rhs`, and then `rhs` goes out of scope. When `operator=`, that implicitly cleans the original resources of `this` (via the destructor, which we didn't have to duplicate). Self-assignment works too - it's less efficient with copy-and-swap (involves an extra allocation and deallocation), but if that's the unlikely scenario, we don't slow down the typical use case to account for it.

Version ≥ C++11

The above formulation works as-is already for move assignment.

```
p1 = std::move(p2);
```

Here, we move-construct `rhs` from `p2`, and all the rest is just as valid. If a class is movable but not copyable, there is no need to delete the copy-assignment, since this assignment operator will simply be ill-formed due to the deleted copy constructor.

Section 40.4: Implicit Move and Copy

Bear in mind that declaring a destructor inhibits the compiler from generating implicit move constructors and move assignment operators. If you declare a destructor, remember to also add appropriate definitions for the move operations.

Furthermore, declaring move operations will suppress the generation of copy operations, so these should also be added (if the objects of this class are required to have copy semantics).

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    //    compiler won't generate these unless we tell it to
    //    because we declared a destructor
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    //    declaring move operations will suppress generation
    //    of copy operations unless we explicitly re-enable them
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

Chapter 41: Non-Static Member Functions

Section 41.1: Non-static Member Functions

A `class` or `struct` can have member functions as well as member variables. These functions have syntax mostly similar to standalone functions, and can be defined either inside or outside the class definition; if defined outside the class definition, the function's name is prefixed with the class' name and the scope (`::`) operator.

```
class CL {
  public:
    void  definedInside() {}
    void definedOutside();
};
void CL::definedOutside() {}
```

These functions are called on an instance (or reference to an instance) of the class with the dot (`.`) operator, or a pointer to an instance with the arrow (`->`) operator, and each call is tied to the instance the function was called on; when a member function is called on an instance, it has access to all of that instance's fields (through the `this` pointer), but can only access other instances' fields if those instances are supplied as parameters.

```
struct ST {
    ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) { }

    int get_i() const { return i; }
    bool compare_i(const ST& other) const { return (i == other.i); }

  private:
    std::string s;
    int i;
};
ST st1;
ST st2("Species", 8472);

int  i = st1.get_i(); // Can access st1.i, but not st2.i.
bool b = st1.compare_i(st2); // Can access st1 & st2.
```

These functions are allowed to access member variables and/or other member functions, regardless of either the variable or function's access modifiers. They can also be written out-of-order, accessing member variables and/or calling member functions declared before them, as the entire class definition must be parsed before the compiler can begin to compile a class.

```
class Access {
  public:
    Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}

    int i;
    int get_k() const { return k; }
    bool private_no_more() const { return i_be_private(); }
  protected:
    int j;
    int get_i() const { return i; }
  private:
    int k;
    int get_j() const { return j; }
    bool i_be_private() const { return ((i > j) && (k < j)); }
};
```

Section 41.2: Encapsulation

A common use of member functions is for encapsulation, using an *accessor* (commonly known as a getter) and a *mutator* (commonly known as a setter) instead of accessing fields directly.

```
class Encapsulator {
    int encapsulated;

  public:
    int  get_encapsulated() const { return encapsulated; }
    void set_encapsulated(int e)  { encapsulated = e; }

    void some_func() {
        do_something_with(encapsulated);
    }
};
```

Inside the class, `encapsulated` can be freely accessed by any non-static member function; outside the class, access to it is regulated by member functions, using `get_encapsulated()` to read it and `set_encapsulated()` to modify it. This prevents unintentional modifications to the variable, as separate functions are used to read and write it. [There are many discussions on whether getters and setters provide or break encapsulation, with good arguments for both claims; such heated debate is outside the scope of this example.]

Section 41.3: Name Hiding & Importing

When a base class provides a set of overloaded functions, and a derived class adds another overload to the set, this hides all of the overloads provided by the base class.

```
struct HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

struct HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1);    // Output:  int
hb.f(true); // Output:  bool
hb.f(s);    // Output:  std::string;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  float
hd.f(true); // Output:  float
hd.f(s);    // Error: Can't convert from std::string to float.
```

This is due to name resolution rules: During name lookup, once the correct name is found, we stop looking, even if we clearly haven't found the correct *version* of the entity with that name (such as with `hd.f(s)`); due to this, overloading the function in the derived class prevents name lookup from discovering the overloads in the base class. To avoid this, a using-declaration can be used to "import" names from the base class into the derived class, so that they will be available during name lookup.

```
struct HidingDerived : HiddenBase {
     // All members named HiddenBase::f shall be considered members of HidingDerived for lookup.
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  int
hd.f(true); // Output:  bool
hd.f(s);    // Output:  std::string
```

If a derived class imports names with a using-declaration, but also declares functions with the same signature as functions in the base class, the base class functions will silently be overridden or hidden.

```
struct NamesHidden {
    virtual void hide_me()      {}
    virtual void hide_me(float) {}
    void hide_me(int)           {}
    void hide_me(bool)          {}
};

struct NameHider : NamesHidden {
    using NamesHidden::hide_me;

    void hide_me()    {} // Overrides NamesHidden::hide_me().
    void hide_me(int) {} // Hides NamesHidden::hide_me(int).
};
```

A using-declaration can also be used to change access modifiers, provided the imported entity was `public` or `protected` in the base class.

```
struct ProMem {
  protected:
    void func() {}
};

struct BecomesPub : ProMem {
    using ProMem::func;
};

// ...

ProMem pm;
BecomesPub bp;

pm.func(); // Error: protected.
bp.func(); // Good.
```

Similarly, if we explicitly want to call a member function from a specific class in the inheritance hierarchy, we can qualify the function name when calling the function, specifying that class by name.

```
struct One {
    virtual void f() { std::cout << "One." << std::endl; }
};

struct Two : One {
    void f() override {
        One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

struct Three : Two {
    void f() override {
        Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();      // Normal syntax.
t.Two::f(); // Calls version of f() defined in Two.
t.One::f(); // Calls version of f() defined in One.
```

Section 41.4: Virtual Member Functions

Member functions can also be declared `virtual`. In this case, if called on a pointer or reference to an instance, they will not be accessed directly; rather, they will look up the function in the virtual function table (a list of pointers-to-member-functions for virtual functions, more commonly known as the `vtable` or `vftable`), and use that to call the version appropriate for the instance's dynamic (actual) type. If the function is called directly, from a variable of a class, no lookup is performed.

```
struct Base {
    virtual void func() { std::cout << "In Base." << std::endl; }
};

struct Derived : Base {
    void func() override { std::cout << "In Derived." << std::endl; }
};

void slicer(Base x) { x.func(); }

// ...

Base b;
Derived d;

Base *pb = &b, *pd = &d; // Pointers.
Base &rb = b, &rd = d;   // References.

b.func();   // Output:  In Base.
d.func();   // Output:  In Derived.

pb->func(); // Output:  In Base.
pd->func(); // Output:  In Derived.

rb.func();  // Output:  In Base.
rd.func();  // Output:  In Derived.
```

```
slicer(b);  // Output:  In Base.
slicer(d);  // Output:  In Base.
```

Note that while `pd` is `Base*`, and `rd` is a `Base&`, calling `func()` on either of the two calls `Derived::func()` instead of `Base::func()`; this is because the `vtable` for `Derived` updates the `Base::func()` entry to instead point to `Derived::func()`. Conversely, note how passing an instance to `slicer()` always results in `Base::func()` being called, even when the passed instance is a `Derived`; this is because of something known as *data slicing*, where passing a `Derived` instance into a `Base` parameter by value renders the portion of the `Derived` instance that isn't a `Base` instance inaccessible.

When a member function is defined as virtual, all derived class member functions with the same signature override it, regardless of whether the overriding function is specified as `virtual` or not. This can make derived classes harder for programmers to parse, however, as there's no indication as to which function(s) is/are `virtual`.

```
struct B {
    virtual void f() {}
};

struct D : B {
    void f() {} // Implicitly virtual, overrides B::f.
                //  You'd have to check B to know that, though.
};
```

Note, however, that a derived function only overrides a base function if their signatures match; even if a derived function is explicitly declared `virtual`, it will instead create a new virtual function if the signatures are mismatched.

```
struct BadB {
    virtual void f() {}
};

struct BadD : BadB {
    virtual void f(int i) {} // Does NOT override BadB::f.
};
```

Version ≥ C++11

As of C++11, intent to override can be made explicit with the context-sensitive keyword `override`. This tells the compiler that the programmer expects it to override a base class function, which causes the compiler to omit an error if it *doesn't* override anything.

```
struct CPP11B {
    virtual void f() {}
};

struct CPP11D : CPP11B {
    void f() override {}
    void f(int i) override {} // Error: Doesn't actually override anything.
};
```

This also has the benefit of telling programmers that the function is both virtual, and also declared in at least one base class, which can make complex classes easier to parse.

When a function is declared `virtual`, and defined outside the class definition, the `virtual` specifier must be included in the function declaration, and not repeated in the definition.

Version ≥ C++11

This also holds true for override.

```
struct VB {
    virtual void f(); // "virtual" goes here.
    void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

If a base class overloads a virtual function, only overloads that are explicitly specified as virtual will be virtual.

```
struct BOverload {
    virtual void func() {}
    void func(int) {}
};

struct DOverload : BOverload {
    void func() override {}
    void func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // Calls DOverload::func().
bo->func(1); // Calls BOverload::func(int).
```

For more information, see the relevant topic.

Section 41.5: Const Correctness

One of the primary uses for this cv-qualifiers is *const correctness*. This is the practice of guaranteeing that only accesses that *need* to modify an object are *able* to modify the object, and that any (member or non-member) function that doesn't need to modify an object doesn't have write access to that object (whether directly or indirectly). This prevents unintentional modifications, making code less errorprone. It also allows any function that doesn't need to modify state to be able to take either a const or non-const object, without needing to rewrite or overload the function.

const correctness, due to its nature, starts at the bottom up: Any class member function that doesn't need to change state is declared as const, so that it can be called on const instances. This, in turn, allows passed-by-reference parameters to be declared const when they don't need to be modified, which allows functions to take either const or non-const objects without complaining, and const-ness can propagate outwards in this manner. Due to this, getters are frequently const, as are any other functions that don't need to modify logical state.

```
class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(const Field& f) : fld(f) {}     // Modifies.

    const Field& get_field()       { return fld; } // Doesn't modify; should be const.
    void set_field(const Field& f) { fld = f; }    // Modifies.

    void do_something(int i) {                     // Modifies.
        fld.insert_value(i);
    }
    void do_nothing()        { }                   // Doesn't modify; should be const.
};
```

```
class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f) : fld(f) {}       // Not const: Modifies.

    const Field& get_field() const { return fld; } // const: Doesn't modify.
    void set_field(const Field& f) { fld = f; }    // Not const: Modifies.

    void do_something(int i) {                     // Not const: Modifies.
        fld.insert_value(i);
    }
    void do_nothing() const  { }                   // const: Doesn't modify.
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// Now, let's read it...
Field f = i_cant_do_anything.get_field();
  // Error: Loses cv-qualifiers, get_field() isn't const.
i_cant_do_anything.do_nothing();
  // Error: Same as above.
// Oops.

const ConstCorrect but_i_can(make_me_a_field());
// Now, let's read it...
Field f = but_i_can.get_field(); // Good.
but_i_can.do_nothing();          // Good.
```

As illustrated by the comments on `ConstIncorrect` and `ConstCorrect`, properly cv-qualifying functions also serves as documentation. If a class is `const` correct, any function that isn't `const` can safely be assumed to change state, and any function that is `const` can safely be assumed not to change state.

Chapter 42: Constant class member functions

Section 42.1: constant member function

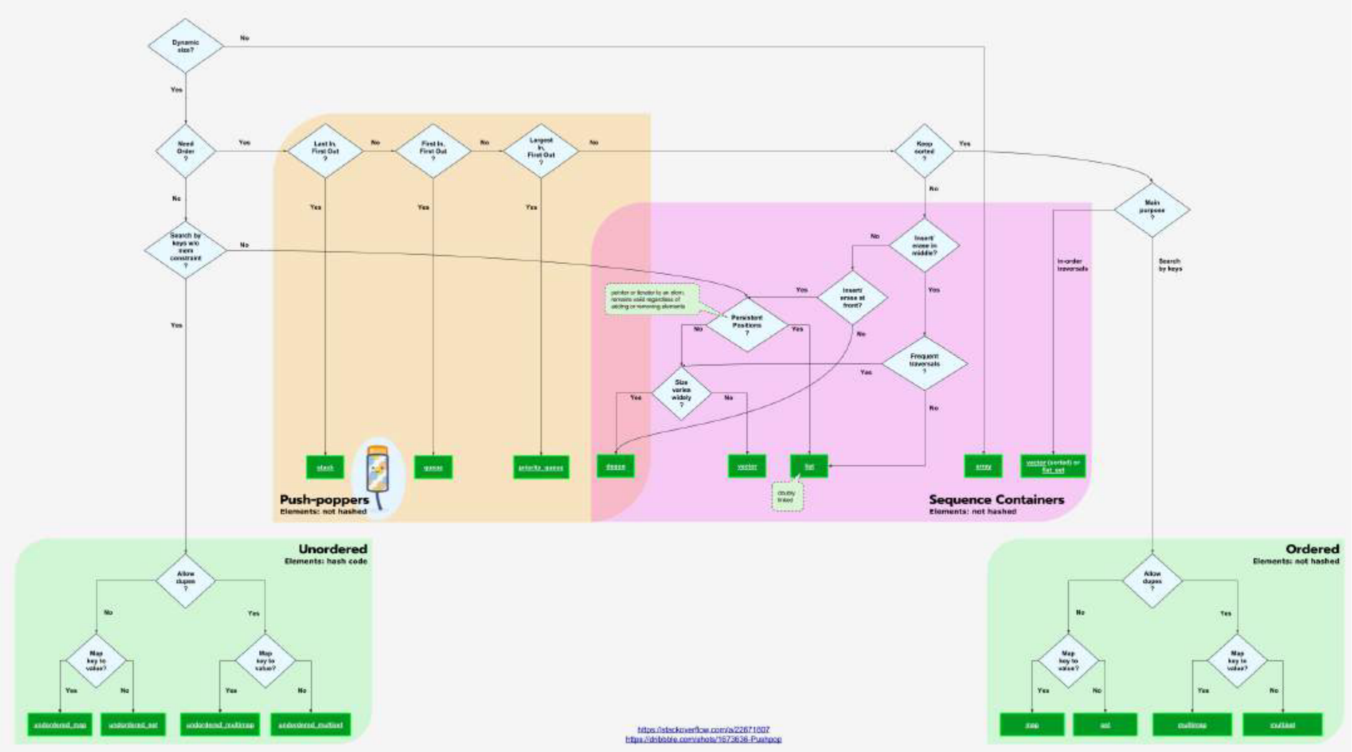
```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
        (*mapOfStrings)[key] = value;             // This works? Yes it does.
        delete mapOfStrings;                      // This also works
        mapOfStrings = new map<string, string>(); // This * does * not work
    }

    void refresh() {
        delete mapOfStrings;
        mapOfStrings = new map<string, string>(); // Works as refresh is non const function
    }

    void getEntry(string const & key) const {
        cout << mapOfStrings->at(key);
    }
};

int main(int argc, char* argv[]) {

    A var;
    var.insertEntry("abc", "abcValue");
    var.getEntry("abc");
    getchar();
    return 0;
}
```

Chapter 43: C++ Containers

C++ containers store a collection of elements. Containers include vectors, lists, maps, etc. Using Templates, C++ containers contain collections of primitives (e.g. ints) or custom classes (e.g. MyClass).

Section 43.1: C++ Containers Flowchart

Choosing which C++ Container to use can be tricky, so here's a simple flowchart to help decide which Container is right for the job.



This flowchart was based on Mikael Persson's post. This little graphic in the flowchart is from Megan Hopkins

Chapter 44: Namespaces

Used to prevent name collisions when using multiple libraries, a namespace is a declarative prefix for functions, classes, types, etc.

Section 44.1: What are namespaces?

A C++ namespace is a collection of C++ entities (functions, classes, variables), whose names are prefixed by the name of the namespace. When writing code within a namespace, named entities belonging to that namespace need not be prefixed with the namespace name, but entities outside of it must use the fully qualified name. The fully qualified name has the format **`<namespace>::<entity>`**. Example:

```
namespace Example
{
  const int test = 5;

  const int test2 = test + 12; //Works within `Example` namespace
}

const int test3 = test + 3; //Fails; `test` not found outside of namespace.

const int test3 = Example::test + 3; //Works; fully qualified name used.
```

Namespaces are useful for grouping related definitions together. Take the analogy of a shopping mall. Generally a shopping mall is split up into several stores, each store selling items from a specific category. One store might sell electronics, while another store might sell shoes. These logical separations in store types help the shoppers find the items they're looking for. Namespaces help c++ programmers, like shoppers, find the functions, classes, and variables they're looking for by organizing them in a logical manner. Example:

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // Description of a Headphone (color, brand, model number, etc.)
    };
    class Television
    {
        // Description of a Television (color, brand, model number, etc.)
    };
}

namespace Shoes
{
    int TotalStock;
    class Sandal
    {
        // Description of a Sandal (color, brand, model number, etc.)
    };
    class Slipper
    {
        // Description of a Slipper (color, brand, model number, etc.)
    };
}
```

There is a single namespace predefined, which is the global namespace that has no name, but can be denoted by `::`. Example:

```
void bar() {
    // defined in global namespace
}
namespace foo {
    void bar() {
        // defined in namespace foo
    }
    void barbar() {
        bar();   // calls foo::bar()
        ::bar(); // calls bar() defined in global namespace
    }
}
```

Section 44.2: Argument Dependent Lookup

When calling a function without an explicit namespace qualifier, the compiler can choose to call a function within a namespace if one of the parameter types to that function is also in that namespace. This is called "Argument Dependent Lookup", or ADL:

```
namespace Test
{
  int call(int i);

  class SomeClass {...};

  int call_too(const SomeClass &data);
}

call(5); //Fails. Not a qualified function name.

Test::SomeClass data;

call_too(data); //Succeeds
```

`call` fails because none of its parameter types come from the `Test` namespace. `call_too` works because `SomeClass` is a member of `Test` and therefore it qualifies for ADL rules.

When does ADL not occur

ADL does not occur if normal unqualified lookup finds a class member, a function that has been declared at block scope, or something that is not of function type. For example:

```
void foo();
namespace N {
    struct X {};
    void foo(X ) { std::cout << '1'; }
    void qux(X ) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {
        foo(N::X{}); // error: ADL is disabled and C::foo() takes no arguments
    }
};

void bar() {
    extern void foo(); // redeclares ::foo
    foo(N::X{});       // error: ADL is disabled and ::foo() doesn't take any arguments
}

int qux;

void baz() {
    qux(N::X{}); // error: variable declaration disables ADL for "qux"
}
```

Section 44.3: Extending namespaces

A useful feature of `namespace`s is that you can expand them (add members to it).

```
namespace Foo
{
    void bar() {}
}

//some other stuff

namespace Foo
{
    void bar2() {}
}
```

Section 44.4: Using directive

The keyword 'using' has three flavors. Combined with keyword 'namespace' you write a 'using directive':

If you don't want to write `Foo::` in front of every stuff in the namespace `Foo`, you can use `using namespace Foo;` to import every single thing out of `Foo`.

```
namespace Foo
{
    void bar() {}
    void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;
bar(); //OK
baz(); //OK
```

It is also possible to import selected entities in a namespace rather than the whole namespace:

```
using Foo::bar;
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported
```

A word of caution: `using namespace`s in header files is seen as bad style in most cases. If this is done, the namespace is imported in *every* file that includes the header. Since there is no way of "un-`using`" a namespace, this can lead to namespace pollution (more or unexpected symbols in the global namespace) or, worse, conflicts. See this example for an illustration of the problem:

```
/***** foo.h *****/
```

```
namespace Foo
{
    class C;
}

/***** bar.h *****/
namespace Bar
{
    class C;
}

/***** baz.h *****/
#include "foo.h"
using namespace Foo;

/***** main.cpp *****/
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // error: Ambiguity between Bar::C and Foo::C
```

A *using-directive* cannot occur at class scope.

Section 44.5: Making namespaces

Creating a namespace is really easy:

```
//Creates namespace foo
namespace Foo
{
    //Declares function bar in namespace foo
    void bar() {}
}
```

To call `bar`, you have to specify the namespace first, followed by the scope resolution operator `::`:

```
Foo::bar();
```

It is allowed to create one namespace in another, for example:

```
namespace A
{
    namespace B
    {
        namespace C
        {
            void bar() {}
        }
    }
}
```

Version ≥ C++17

The above code could be simplified to the following:

```
namespace A::B::C
{
    void bar() {}
}
```

Section 44.6: Unnamed/anonymous namespaces

An unnamed namespace can be used to ensure names have internal linkage (can only be referred to by the current translation unit). Such a namespace is defined in the same way as any other namespace, but without the name:

```
namespace {
    int foo = 42;
}
```

`foo` is only visible in the translation unit in which it appears.

It is recommended to never use unnamed namespaces in header files as this gives a version of the content for every translation unit it is included in. This is especially important if you define non-const globals.

```
// foo.h
namespace {
    std::string globalString;
}

// 1.cpp
#include "foo.h" //< Generates unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";

// 2.cpp
#include "foo.h" //< Generates unnamed_namespace{2.cpp}::globalString ...

std::cout << globalString; //< Will always print the empty string
```

Section 44.7: Compact nested namespaces

Version ≥ C++17

```
namespace a {
  namespace b {
    template<class T>
    struct qualifies : std::false_type {};
  }
}

namespace other {
  struct bob {};
}

namespace a::b {
  template<>
  struct qualifies<::other::bob> : std::true_type {};
}
```

You can enter both the `a` and `b` namespaces in one step with `namespace a::b` starting in C++17.

Section 44.8: Namespace alias

A namespace can be given an alias (*i.e.,* another name for the same namespace) using the `namespace` *identifier* = syntax. Members of the aliased namespace can be accessed by qualifying them with the name of the alias. In the following example, the nested namespace `AReallyLongName::AnotherReallyLongName` is inconvenient to type, so the function `qux` locally declares an alias `N`. Members of that namespace can then be accessed simply using `N::`.

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int foo();
        int bar();
        void baz(int x, int y);
    }
}
void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::baz(N::foo(), N::bar());
}
```

Section 44.9: Inline namespace

Version ≥ C++11

`inline namespace` includes the content of the inlined namespace in the enclosing namespace, so

```
namespace Outer
{
    inline namespace Inner
    {
        void foo();
    }
}
```

is mostly equivalent to

```
namespace Outer
{

    namespace Inner
    {
        void foo();
    }

    using Inner::foo;
}
```

but element from `Outer::Inner::` and those associated into `Outer::` are identical.

So following is equivalent

```
Outer::foo();
Outer::Inner::foo();
```

The alternative `using namespace Inner;` would not be equivalent for some tricky parts as template specialization:

For

```
#include <outer.h> // See below

class MyCustomType;
namespace Outer
{
    template <>
    void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- The inline namespace allows the specialization of `Outer::foo`

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
    inline namespace Inner
    {
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
}
```

- Whereas the `using namespace` doesn't allow the specialization of `Outer::foo`

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
    namespace Inner
    {
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
    using namespace Inner;
    // Specialization of `Outer::foo` is not possible
    // it should be `Outer::Inner::foo`.
}
```

Inline namespace is a way to allow several version to cohabit and defaulting to the `inline` one

```
namespace MyNamespace
{
    // Inline the last version
    inline namespace Version2
    {
        void foo(); // New version
        void bar();
    }

    namespace Version1 // The old one
    {
        void foo();
    }

}
```

And with usage

```
MyNamespace::Version1::foo(); // old version
MyNamespace::Version2::foo(); // new version
MyNamespace::foo();           // default version : MyNamespace::Version1::foo();
```

Section 44.10: Aliasing a long namespace

This is usually used for renaming or shortening long namespace references such referring to components of a library.

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;


//    Both Type declarations are equivalent
boost::multiprecision::Number X   //    Writing the full namespace path, longer
Name1::Number Y                   //    using the name alias, shorter
```

Section 44.11: Alias Declaration scope

Alias Declaration are affected by preceding *using* statements

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}


using namespace boost;

//   Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

However, it is easier to get confused over which namespace you are aliasing when you have something like this:

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace numeric;
using namespace boost;
```

```
//    Not recommended as
//    its not explicitly clear whether Name1 refers to
//    numeric::multiprecision or boost::multiprecision
namespace Name1 = multiprecision;

//    For clarity, its recommended to use absolute paths
//    instead
namespace Name2 = numeric::multiprecision;
namespace Name3 = boost::multiprecision;
```

Chapter 45: Header Files

Section 45.1: Basic Example

The following example will contain a block of code that is meant to be split into several source files, as denoted by `// filename` comments.

Source Files

```
// my_function.h

/* Note how this header contains only a declaration of a function.
 * Header functions usually do not define implementations for declarations
 * unless code must be further processed at compile time, as in templates.
 */

/* Also, usually header files include preprocessor guards so that every header
 * is never included twice.
 *
 * The guard is implemented by checking if a header-file unique preprocessor
 * token is defined, and only including the header if it hasn't been included
 * once before.
 */
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value and my_function() will be
// recognized as the same constructs if this header is included by different files.
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* Note how the corresponding source file for the header includes the interface
 * defined in the header so that the compiler is aware of what the source file is
 * implementing.
 *
 * In this case, the source file requires knowledge of the global constant
 * global_value only defined in my_function.h. Without inclusion of the header
 * file, this source file would not compile.
 */
#include "my_function.h" // or #include "my_function.hpp"
int my_function() {
  return global_value; // return 42;
}
```

Header files are then included by other source files that want to use the functionality defined by the header interface, but don't require knowledge of its implementation (thus, reducing code coupling). The following program makes use of the header `my_function.h` as defined above:

```
// main.cpp

#include <iostream>       // A C++ Standard Library header.
#include "my_function.h"  // A personal header

int main(int argc, char** argv) {
  std::cout << my_function() << std::endl;
  return 0;
}
```

The Compilation Process

Since header files are often part of a compilation process workflow, a typical compilation process making use of the header/source file convention will usually do the following.

Assuming that the header file and source code file is already in the same directory, a programmer would execute the following commands:

```
g++ -c my_function.cpp       # Compiles the source file my_function.cpp
                             # --> object file my_function.o

g++ main.cpp my_function.o   # Links the object file containing the
                             # implementation of int my_function()
                             # to the compiled, object version of main.cpp
                             # and then produces the final executable a.out
```

Alternatively, if one wishes to compile `main.cpp` to an object file first, and then link only object files together as the final step:

```
g++ -c my_function.cpp
g++ -c main.cpp

g++ main.o my_function.o
```

Section 45.2: Templates in Header Files

Templates require compile-time generation of code: a templated function, for example, will be effectively turned into multiple distinct functions once a templated function is parameterized by use in source code.

This means that template function, member function, and class definitions cannot be delegated to a separate source code file, as any code that will use any templated construct requires knowledge of its definition to generally generate any derivative code.

Thus, templated code, if put in headers, must also contain its definition. An example of this is below:

```
// templated_function.h

template <typename T>
T* null_T_pointer() {
  T* type_point = NULL; // or, alternatively, nullptr instead of NULL
                        // for C++11 or later
  return type_point;
}
```

Chapter 46: Using declaration

A `using` declaration introduces a single name into the current scope that was previously declared elsewhere.

Section 46.1: Importing names individually from a namespace

Once `using` is used to introduce the name `cout` from the namespace `std` into the scope of the `main` function, the `std::cout` object can be referred to as `cout` alone.

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!\n";
}
```

Section 46.2: Redeclaring members from a base class to avoid name hiding

If a *using-declaration* occurs at class scope, it is only allowed to redeclare a member of a base class. For example, `using std::cout` is not allowed at class scope.

Often, the name redeclared is one that would otherwise be hidden. For example, in the below code, `d1.foo` only refers to `Derived1::foo(const char*)` and a compilation error will occur. The function `Base::foo(int)` is hidden not considered at all. However, `d2.foo(42)` is fine because the *using-declaration* brings `Base::foo(int)` into the set of entities named `foo` in `Derived2`. Name lookup then finds both `foo`s and overload resolution selects `Base::foo`.

```
struct Base {
    void foo(int);
};
struct Derived1 : Base {
    void foo(const char*);
};
struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};
int main() {
    Derived1 d1;
    d1.foo(42);  // error
    Derived2 d2;
    d2.foo(42);  // OK
}
```

Section 46.3: Inheriting constructors

Version ≥ C++11

As a special case, a *using-declaration* at class scope can refer to the constructors of a direct base class. Those constructors are then *inherited* by the derived class and can be used to initialize the derived class.

```
struct Base {
    Base(int x, const char* s);
};
struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
```

```
struct Derived2 : Base {
    using Base::Base;
};
int main() {
    Derived1 d1(42, "Hello, world");
    Derived2 d2(42, "Hello, world");
}
```

In the above code, both `Derived1` and `Derived2` have constructors that forward the arguments directly to the corresponding constructor of `Base`. `Derived1` performs the forwarding explicitly, while `Derived2`, using the C++11 feature of inheriting constructors, does so implicitly.

Chapter 47: std::string

Strings are objects that represent sequences of characters. The standard `string` class provides a simple, safe and versatile alternative to using explicit arrays of `char`s when dealing with text and other sequences of characters. The C++ `string` class is part of the `std` namespace and was standardized in 1998.

Section 47.1: Tokenize

Listed from least expensive to most expensive at run-time:

1. `std::strtok` is the cheapest standard provided tokenization method, it also allows the delimiter to be modified between tokens, but it incurs 3 difficulties with modern C++:

    - `std::strtok` cannot be used on multiple `strings` at the same time (though some implementations do extend to support this, such as: `strtok_s`)
    - For the same reason `std::strtok` cannot be used on multiple threads simultaneously (this may however be implementation defined, for example: Visual Studio's implementation is thread safe)
    - Calling `std::strtok` modifies the `std::string` it is operating on, so it cannot be used on `const strings`, `const char*`s, or literal strings, to tokenize any of these with `std::strtok` or to operate on a `std::string` who's contents need to be preserved, the input would have to be copied, then the copy could be operated on

    Generally any of these options cost will be hidden in the allocation cost of the tokens, but if the cheapest algorithm is required and `std::strtok`'s difficulties are not overcomable consider a hand-spun solution.

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Live Example

2. The `std::istream_iterator` uses the stream's extraction operator iteratively. If the input `std::string` is white-space delimited this is able to expand on the `std::strtok` option by eliminating its difficulties, allowing inline tokenization thereby supporting the generation of a `const vector<string>`, and by adding support for multiple delimiting white-space character:

```
// String to tokenize
const std::string str("The  quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
                                        std::istream_iterator<std::string>(is),
                                        std::istream_iterator<std::string>());
```

Live Example

3. The `std::regex_token_iterator` uses a `std::regex` to iteratively tokenize. It provides for a more flexible delimiter definition. For example, non-delimited commas and white-space:

Version ≥ C++11

```
// String to tokenize
const std::string str{ "The ,qu\\,ick ,\tbrown, fox" };
const std::regex re{ "\\s*((?:[^\\\\,]|\\\\.)*?)\\s*(?:,|$)" };
// Vector to store tokens
const std::vector<std::string> tokens{
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),
    std::sregex_token_iterator()
};
```

Live Example

See the `regex_token_iterator` Example for more details.

Section 47.2: Conversion to (const) char*

In order to get `const char*` access to the data of a `std::string` you can use the string's `c_str()` member function. Keep in mind that the pointer is only valid as long as the `std::string` object is within scope and remains unchanged, that means that only `const` methods may be called on the object.

Version ≥ C++17

The `data()` member function can be used to obtain a modifiable `char*`, which can be used to manipulate the `std::string` object's data.

Version ≥ C++11

A modifiable `char*` can also be obtained by taking the address of the first character: `&s[0]`. Within C++11, this is guaranteed to yield a well-formed, null-terminated string. Note that `&s[0]` is well-formed even if `s` is empty, whereas `&s.front()` is undefined if `s` is empty.

Version ≥ C++11

```
std::string str("This is a string.");
const char* cstr = str.c_str(); // cstr points to: "This is a string.\0"
const char* data = str.data();  // data points to: "This is a string.\0"
```

```
std::string str("This is a string.");

// Copy the contents of str to untie lifetime from the std::string object
std::unique_ptr<char []> cstr = std::make_unique<char[]>(str.size() + 1);

// Alternative to the line above (no exception safety):
// char* cstr_unsafe = new char[str.size() + 1];

std::copy(str.data(), str.data() + str.size(), cstr);
cstr[str.size()] = '\0'; // A null-terminator needs to be added

// delete[] cstr_unsafe;
std::cout << cstr.get();
```

Section 47.3: Using the std::string_view class

Version ≥ C++17

C++17 introduces `std::string_view`, which is simply a non-owning range of `const char`s, implementable as either a pair of pointers or a pointer and a length. It is a superior parameter type for functions that requires non-modifiable string data. Before C++17, there were three options for this:

```
void foo(std::string const& s);      // pre-C++17, single argument, could incur
```

```
                                            // allocation if caller's data was not in a string
                                            // (e.g. string literal or vector<char> )

void foo(const char* s, size_t len); // pre-C++17, two arguments, have to pass them
                                            // both everywhere

void foo(const char* s);                    // pre-C++17, single argument, but need to call
                                            // strlen()

template <class StringT>
void foo(StringT const& s);                 // pre-C++17, caller can pass arbitrary char data
                                            // provider, but now foo() has to live in a header
```

All of these can be replaced with:

```
void foo(std::string_view s);               // post-C++17, single argument, tighter coupling
                                            // zero copies regardless of how caller is storing
                                            // the data
```

*Note that `std::string_view` **cannot** modify its underlying data.*

`string_view` is useful when you want to avoid unnecessary copies.

It offers a useful subset of the functionality that `std::string` does, although some of the functions behave differently:

```
std::string str = "lllloooonnnngggg sssstttrrriiinnnggg"; //A really long string

//Bad way - 'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//Good way - No copies are created!
std::string_view view = str;

// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';
```

Section 47.4: Conversion to std::wstring

In C++, sequences of characters are represented by specializing the `std::basic_string` class with a native character type. The two major collections defined by the standard library are `std::string` and `std::wstring`:

- `std::string` is built with elements of type `char`
- `std::wstring` is built with elements of type `wchar_t`

To convert between the two types, use `wstring_convert`:

```
#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// conversion
std::wstring str_turned_to_wstr =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);
```

```
std::string wstr_turned_to_str =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

In order to improve usability and/or readability, you can define functions to perform the conversion:

```
#include <string>
#include <codecvt>
#include <locale>

using convert_t = std::codecvt_utf8<wchar_t>;
std::wstring_convert<convert_t, wchar_t> strconverter;

std::string to_string(std::wstring wstr)
{
    return strconverter.to_bytes(wstr);
}

std::wstring to_wstring(std::string str)
{
    return strconverter.from_bytes(str);
}
```

Sample usage:

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

That's certainly more readable than `std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")`.

Please note that `char` and `wchar_t` do not imply encoding, and gives no indication of size in bytes. For instance, `wchar_t` is commonly implemented as a 2-bytes data type and typically contains UTF-16 encoded data under Windows (or UCS-2 in versions prior to Windows 2000) and as a 4-bytes data type encoded using UTF-32 under Linux. This is in contrast with the newer types `char16_t` and `char32_t`, which were introduced in C++11 and are guaranteed to be large enough to hold any UTF16 or UTF32 "character" (or more precisely, *code point*) respectively.

Section 47.5: Lexicographical comparison

Two `std::string`s can be compared lexicographically using the operators `==`, `!=`, `<`, `<=`, `>`, and `>=`:

```
std::string str1 = "Foo";
std::string str2 = "Bar";

assert(!(str1 < str2));
assert(str > str2);
assert(!(str1 <= str2));
assert(str1 >= str2);
assert(!(str1 == str2));
assert(str1 != str2);
```

All these functions use the underlying `std::string::compare()` method to perform the comparison, and return for convenience boolean values. The operation of these functions may be interpreted as follows, regardless of the actual implementation:

- operator`==`:

  If `str1.length() == str2.length()` and each character pair matches, then returns `true`, otherwise returns `false`.

- operator`!=`:

  If `str1.length() != str2.length()` or one character pair doesn't match, returns `true`, otherwise it returns `false`.

- operator`<` or operator`>`:

  Finds the first different character pair, compares them then returns the boolean result.

- operator`<=` or operator`>=`:

  Finds the first different character pair, compares them then returns the boolean result.

Note: The term **character pair** means the corresponding characters in both strings of the same positions. For better understanding, if two example strings are `str1` and `str2`, and their lengths are `n` and `m` respectively, then character pairs of both strings means each `str1[i]` and `str2[i]` pairs where *i = 0, 1, 2, ..., max(n,m)*. If for any *i* where the corresponding character does not exist, that is, when *i* is greater than or equal to `n` or `m`, it would be considered as the lowest value.

Here is an example of using `<`:

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

The steps are as follows:

1. Compare the first characters, `'B' == 'B'` - move on.
2. Compare the second characters, `'a' == 'a'` - move on.
3. Compare the third characters, `'r' == 'r'` - move on.
4. The `str2` range is now exhausted, while the `str1` range still has characters. Thus, `str2 < str1`.

Section 47.6: Trimming characters at start/end

This example requires the headers **`<algorithm>`**, **`<locale>`**, and **`<utility>`**.

Version ≥ C++11

To *trim* a sequence or string means to remove all leading and trailing elements (or characters) matching a certain predicate. We first trim the trailing elements, because it doesn't involve moving any elements, and then trim the leading elements. Note that the generalizations below work for all types of `std::basic_string` (e.g. `std::string` and `std::wstring`), and accidentally also for sequence containers (e.g. `std::vector` and `std::list`).

```
template <typename Sequence, // any basic_string, vector, list etc.
          typename Pred>     // a predicate on the element (character) type
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

Trimming the trailing elements involves finding the *last* element not matching the predicate, and erasing from there on:

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                 seq.rend(),
                                 pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

Trimming the leading elements involves finding the *first* element not matching the predicate and erasing up to there:

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                  seq.end(),
                                  pred);
    seq.erase(seq.begin(), first);
    return seq;
}
```

To specialize the above for trimming whitespace in a `std::string` we can use the `std::isspace()` function as a predicate:

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

Similarly, we can use the `std::iswspace()` function for `std::wstring` etc.

If you wish to create a *new* sequence that is a trimmed copy, then you can use a separate function:

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}
```

Section 47.7: String replacement

Replace by position

To replace a portion of a `std::string` you can use the method `replace` from `std::string`.

`replace` has a lot of useful overloads:

```
//Define string
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";

//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"

//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"

//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
```

Version ≥ C++14

```
//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"
```

```
//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"

//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"

//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
```

Version ≥ C++11

```
//8)
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"
```

Replace occurrences of a string with another string

Replace only the first occurrence of `replace` with `with` in `str`:

```
std::string replaceString(std::string str,
                          const std::string& replace,
                          const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}
```

Replace all occurrence of `replace` with `with` in `str`:

```
std::string replaceStringAll(std::string str,
                             const std::string& replace,
                             const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}
```

Section 47.8: Converting to std::string

`std::ostringstream` can be used to convert any streamable type to a string representation, by inserting the object into a `std::ostringstream` object (with the stream insertion operator `<<`) and then converting the whole `std::ostringstream` to a `std::string`.

For `int` for instance:

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

Writing your own conversion function, the simple:

```
template<class T>
std::string toString(const T& x)
{
  std::ostringstream ss;
  ss << x;
  return ss.str();
}
```

works but isn't suitable for performance critical code.

User-defined classes may implement the stream insertion operator if desired:

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // write a string representation of a to out
    return out;
}
```

Version ≥ C++11

Aside from streams, since C++11 you can also use the `std::to_string` (and `std::to_wstring`) function which is overloaded for all fundamental types and returns the string representation of its parameter.

```
std::string s = to_string(0x12f3);  // after this the string s contains "4851"
```

Section 47.9: Splitting

Use `std::string::substr` to split a string. There are two variants of this member function.

The first takes a *starting position* from which the returned substring should begin. The starting position must be valid in the range `(0, str.length()]`:

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

The second takes a starting position and a total *length* of the new substring. Regardless of the *length*, the substring will never go past the end of the source string:

```
std::string str = "Hello foo, bar and world!";
```

```
std::string newstr = str.substr(15, 3); // "and"
```

Note that you can also call `substr` with no arguments, in this case an exact copy of the string is returned

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

Section 47.10: Accessing a character

There are several ways to extract characters from a `std::string` and each is subtly different.

```
std::string str("Hello world!");
```

operator[](n)

Returns a reference to the character at index n.

`std::string::operator[]` is not bounds-checked and does not throw an exception. The caller is responsible for asserting that the index is within the range of the string:

```
char c = str[6]; // 'w'
```

at(n)

Returns a reference to the character at index n.

`std::string::at` *is* bounds checked, and will throw `std::out_of_range` if the index is not within the range of the string:

```
char c = str.at(7); // 'o'
```

Version ≥ C++11

Note: Both of these examples will result in undefined behavior if the string is empty.

front()

Returns a reference to the first character:

```
char c = str.front(); // 'H'
```

back()

Returns a reference to the last character:

```
char c = str.back(); // '!'
```

Section 47.11: Checking if a string is a prefix of another

Version ≥ C++14

In C++14, this is easily done by `std::mismatch` which returns the first mismatching pair from two ranges:

```
std::string prefix = "foo";
```

```
std::string string = "foobar";

bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),
    string.begin(), string.end()).first == prefix.end();
```

Note that a range-and-a-half version of `mismatch()` existed prior to C++14, but this is unsafe in the case that the second string is the shorter of the two.

Version < C++14

We can still use the range-and-a-half version of `std::mismatch()`, but we need to first check that the first string is at most as big as the second:

```
bool isPrefix = prefix.size() <= string.size() &&
    std::mismatch(prefix.begin(), prefix.end(),
        string.begin(), string.end()).first == prefix.end();
```

Version ≥ C++17

With `std::string_view`, we can write the direct comparison we want without having to worry about allocation overhead or making copies:

```
bool isPrefix(std::string_view prefix, std::string_view full)
{
    return prefix == full.substr(0, prefix.size());
}
```

Section 47.12: Looping through each character

Version ≥ C++11

`std::string` supports iterators, and so you can use a *ranged based* loop to iterate through each character:

```
std::string str = "Hello World!";
for (auto c : str)
    std::cout << c;
```

You can use a "traditional" `for` loop to loop through every character:

```
std::string str = "Hello World!";
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

Section 47.13: Conversion to integers/floating point types

A `std::string` containing a number can be converted into an integer type, or a floating point type, using conversion functions.

Note that all of these functions stop parsing the input string as soon as they encounter a non-numeric character, so `"123abc"` will be converted into `123`.

The `std::ato*` family of functions converts C-style strings (character arrays) to integer or floating-point types:

```
std::string ten = "10";

double num1 = std::atof(ten.c_str());
```

```
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
```

Version ≥ C++11

```
long long num4 = std::atoll(ten.c_str());
```

However, use of these functions is discouraged because they return `0` if they fail to parse the string. This is bad because `0` could also be a valid result, if for example the input string was "0", so it is impossible to determine if the conversion actually failed.

The newer `std::sto*` family of functions convert `std::string`s to integer or floating-point types, and throw exceptions if they could not parse their input. *You should use these functions if possible*:

Version ≥ C++11

```
std::string ten = "10";

int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

Furthermore, these functions also handle octal and hex strings unlike the `std::ato*` family. The second parameter is a pointer to the first unconverted character in the input string (not illustrated here), and the third parameter is the base to use. `0` is automatic detection of octal (starting with `0`) and hex (starting with `0x` or `0X`), and any other value is the base to use

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16);  // Returns 10
long num4 = std::stol(ten_hex);  // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Section 47.14: Concatenation

You can concatenate `std::string`s using the overloaded `+` and `+=` operators. Using the `+` operator:

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

Using the `+=` operator:

```
std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"
```

You can also append C strings, including string literals:

```
std::string hello = "Hello";
std::string world = "world";
```

```
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

You can also use `push_back()` to push back individual `char`s:

```
std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"
```

There is also `append()`, which is pretty much like `+=`:

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

Section 47.15: Converting between character encodings

Converting between encodings is easy with C++11 and most compilers are able to deal with it in a cross-platform manner through **`<codecvt>`** and **`<locale>`** headers.

```
#include <iostream>
#include <codecvt>
#include <locale>
#include <string>
using namespace std;

int main() {
    // converts between wstring and utf8 string
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // converts between u16string and utf8 string
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    u16string u16str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(u16str);
    u16string u16str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
}
```

Mind that Visual Studio 2015 provides supports for these conversion but a bug in their library implementation requires to use a different template for `wstring_convert` when dealing with `char16_t`:

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
  std::basic_string<utf16_char> tmp;
  tmp.resize(utf16.length());
  std::copy(utf16.begin(), utf16.end(), tmp.begin());
  utf8 = conv_utf8_utf16.to_bytes(tmp);
}
```

```
void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
  std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
  utf16.clear();
  utf16.resize(tmp.length());
  std::copy(tmp.begin(), tmp.end(), utf16.begin());
}
```

Section 47.16: Finding character(s) in a string

To find a character or another string, you can use `std::string::find`. It returns the position of the first character of the first match. If no matches were found, the function returns `std::string::npos`

```
std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)
    std::cout << "Found at position: " << it << '\n';
else
    std::cout << "Not found!\n";
```

> Found at position: 21

The search opportunities are further expanded by the following functions:

```
find_first_of     // Find first occurrence of characters
find_first_not_of // Find first absence of characters
find_last_of      // Find last occurrence of characters
find_last_not_of  // Find last absence of characters
```

These functions can allow you to search for characters from the end of the string, as well as find the negative case (ie. characters that are not in the string). Here is an example:

```
std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << '\n';
```

> Found at position: 6

Note: Be aware that the above functions do not search for substrings, but rather for characters contained in the search string. In this case, the last occurrence of `'g'` was found at position `6` (the other characters weren't found).

Chapter 48: std::array

Parameter	Definition
`class T`	Specifies the data type of array members
`std::size_t N`	Specifies the number of members in the array

Section 48.1: Initializing an std::array

Initializing `std::array<T, N>`, where `T` is a scalar type and `N` is the number of elements of type `T`

If `T` is a scalar type, `std::array` can be initialized in the following ways:

```
// 1) Using aggregate-initialization
std::array<int, 3> a{ 0, 1, 2 };
// or equivalently
std::array<int, 3> a = { 0, 1, 2 };

// 2) Using the copy constructor
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> a2(a);
// or equivalently
std::array<int, 3> a2 = a;

// 3) Using the move constructor
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Initializing `std::array<T, N>`, where `T` is a non-scalar type and `N` is the number of elements of type `T`

If `T` is a non-scalar type `std::array` can be initialized in the following ways:

```
struct A { int values[3]; }; // An aggregate type

// 1) Using aggregate initialization with brace elision
// It works only if T is an aggregate type!
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
// or equivalently
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };

// 2) Using aggregate initialization with brace initialization of sub-elements
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// or equivalently
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };

// 3)
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// or equivalently
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};

// 4) Using the copy constructor
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// or equivalently
std::array<A, 2> a2 = a;

// 5) Using the move constructor
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Section 48.2: Element access

1. `at(pos)`

Returns a reference to the element at position `pos` with bounds checking. If `pos` is not within the range of the container, an exception of type `std::out_of_range` is thrown.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // read values
    int a = arr.at(0); // a is now 2
    int b = arr.at(1); // b is now 4
    int c = arr.at(2); // c is now 6

    return 0;
}
```

2) `operator[pos]`

Returns a reference to the element at position `pos` without bounds checking. If `pos` is not within the range of the container, a runtime *segmentation violation* error can occur. This method provides element access equivalent to classic arrays and thereof more efficient than `at(pos)`.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // read values
    int a = arr[0]; // a is now 2
    int b = arr[1]; // b is now 4
    int c = arr[2]; // c is now 6

    return 0;
}
```

3) `std::get<pos>`

This **non-member** function returns a reference to the element at **compile-time constant** position `pos` without bounds checking. If `pos` is not within the range of the container, a runtime *segmentation violation* error can occur.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // read values
    int a = std::get<0>(arr); // a is now 2
    int b = std::get<1>(arr); // b is now 4
    int c = std::get<2>(arr); // c is now 6

    return 0;
}
```

4) `front()`

Returns a reference to the first element in container. Calling `front()` on an empty container is undefined.

The complexity is constant O(1).

Note: For a container c, the expression `c.front()` is equivalent to `*c.begin()`.

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a is now 2

    return 0;
}
```

5) `back()`

Returns reference to the last element in the container. Calling `back()` on an empty container is undefined.

The complexity is constant O(1).

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6) `data()`

Returns pointer to the underlying array serving as element storage. The pointer is such that `range [data(); data() + size())` is always a valid range, even if the container is empty (`data()` is not dereferenceable in that case).

The complexity is constant O(1).

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

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // copy cstr to arr

    std::cout << arr.data(); // outputs: Test string

    return 0;
}
```

Section 48.3: Iterating through the Array

`std::array` being a STL container, can use range-based for loop similar to other containers like `vector`

```
int main() {
     std::array<int, 3> arr = { 1, 2, 3 };
     for (auto i : arr)
         cout << i << '\n';
}
```

Section 48.4: Checking size of the Array

One of the main advantage of `std::array` as compared to `C` style array is that we can check the size of the array using `size()` member function

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

Section 48.5: Changing all array elements at once

The member function `fill()` can be used on `std::array` for changing the values at once post initialization

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

Chapter 49: std::vector

A vector is a dynamic array with automatically handled storage. The elements in a vector can be accessed just as efficiently as those in an array with the advantage being that vectors can dynamically change in size.

In terms of storage the vector data is (usually) placed in dynamically allocated memory thus requiring some minor overhead; conversely `C-arrays` and `std::array` use automatic storage relative to the declared location and thus do not have any overhead.

Section 49.1: Accessing Elements

There are two primary ways of accessing elements in a `std::vector`

- index-based access
- iterators

Index-based access:

This can be done either with the subscript operator `[]`, or the member function `at()`.

Both return a reference to the element at the respective position in the `std::vector` (unless it's a `vector<bool>`), so that it can be read as well as modified (if the vector is not `const`).

`[]` and `at()` differ in that `[]` is not guaranteed to perform any bounds checking, while `at()` does. Accessing elements where `index < 0` or `index >= size` is undefined behavior for `[]`, while `at()` throws a `std::out_of_range` exception.

Note: The examples below use C++11-style initialization for clarity, but the operators can be used with all versions (unless marked C++11).

Version ≥ C++11

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

```
// using []
int a = v[1];    // a is 2
v[1] = 4;        // v now contains { 1, 4, 3 }

// using at()
int b = v.at(2); // b is 3
v.at(2) = 5;     // v now contains { 1, 4, 5 }
int c = v.at(3); // throws std::out_of_range exception
```

Because the `at()` method performs bounds checking and can throw exceptions, it is slower than `[]`. This makes `[]` preferred code where the semantics of the operation guarantee that the index is in bounds. In any case, accesses to elements of vectors are done in constant time. That means accessing to the first element of the vector has the same cost (in time) of accessing the second element, the third element and so on.

For example, consider this loop

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

Here we know that the index variable `i` is always in bounds, so it would be a waste of CPU cycles to check that `i` is in bounds for every call to `operator[]`.

The front() and back() member functions allow easy reference access to the first and last element of the vector, respectively. These positions are frequently used, and the special accessors can be more readable than their alternatives using []:

```
std::vector<int> v{ 4, 5, 6 }; // In pre-C++11 this is more verbose

int a = v.front();   // a is 4, v.front() is equivalent to v[0]
v.front() = 3;       // v now contains {3, 5, 6}
int b = v.back();    // b is 6, v.back() is equivalent to v[v.size() - 1]
v.back() = 7;        // v now contains {3, 5, 7}
```

Note: It is undefined behavior to invoke `front()` or `back()` on an empty vector. You need to check that the container is not empty using the empty() member function (which checks if the container is empty) before calling `front()` or `back()`. A simple example of the use of 'empty()' to test for an empty vector follows:

```
int main ()
{
  std::vector<int> v;
  int sum (0);

  for (int i=1;i<=10;i++) v.push_back(i);//create and initialize the vector

  while (!v.empty())//loop through until the vector tests to be empty
  {
     sum += v.back();//keep a running total
     v.pop_back();//pop out the element which removes it from the vector
  }

  std::cout << "total: " << sum << '\n';//output the total to the user

  return 0;
}
```

The example above creates a vector with a sequence of numbers from 1 to 10. Then it pops the elements of the vector out until the vector is empty (using 'empty()') to prevent undefined behavior. Then the sum of the numbers in the vector is calculated and displayed to the user.

Version ≥ C++11

The data() method returns a pointer to the raw memory used by the `std::vector` to internally store its elements. This is most often used when passing the vector data to legacy code that expects a C-style array.

```
std::vector<int> v{ 1, 2, 3, 4 }; // v contains {1, 2, 3, 4}
int* p = v.data(); // p points to 1
*p = 4;            // v now contains {4, 2, 3, 4}
++p;               // p points to 2
*p = 3;            // v now contains {4, 3, 3, 4}
p[1] = 2;          // v now contains {4, 3, 2, 4}
*(p + 2) = 1;      // v now contains {4, 3, 2, 1}
```

Version < C++11

Before C++11, the `data()` method can be simulated by calling `front()` and taking the address of the returned value:

```
std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]
```

This works because vectors are always guaranteed to store their elements in contiguous memory locations, assuming the contents of the vector doesn't override unary `operator&`. If it does, you'll have to re-implement `std::addressof` in pre-C++11. It also assumes that the vector isn't empty.

Iterators:

Iterators are explained in more detail in the example "Iterating over `std::vector`" and the article Iterators. In short, they act similarly to pointers to the elements of the vector:

Version ≥ C++11

```
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it;        // i is 4
++it;
i = *it;            // i is 5
*it = 6;            // v contains { 4, 6, 6 }
auto e = v.end();   // e points to the element after the end of v. It can be
                    // used to check whether an iterator reached the end of the vector:
++it;
it == v.end();      // false, it points to the element at position 2 (with value 6)
++it;
it == v.end();      // true
```

It is consistent with the standard that a `std::vector<T>`'s iterators actually *be* `T*`s, but most standard libraries do not do this. Not doing this both improves error messages, catches non-portable code, and can be used to instrument the iterators with debugging checks in non-release builds. Then, in release builds, the class wrapping around the underlying pointer is optimized away.

You can persist a reference or a pointer to an element of a vector for indirect access. These references or pointers to elements in the `vector` remain stable and access remains defined unless you add/remove elements at or before the element in the `vector`, or you cause the `vector` capacity to change. This is the same as the rule for invalidating iterators.

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;     // p points to 2
v.insert(v.begin(), 0);    // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;          // p points to 1
v.reserve(10);             // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;          // p points to 1
v.erase(v.begin());        // p is now invalid, accessing *p is a undefined behavior.
```

Section 49.2: Initializing a std::vector

A `std::vector` can be initialized in several ways while declaring it:

Version ≥ C++11

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

// Different from std::vector<int> v(3, 6)
std::vector<int> v{ 3, 6 };     // v becomes {3, 6}
```

```
// Different from std::vector<int> v{3, 6} in C++11
std::vector<int> v(3, 6);  // v becomes {6, 6, 6}

std::vector<int> v(4);     // v becomes {0, 0, 0, 0}
```

A vector can be initialized from another container in several ways:

Copy construction (from another vector only), which copies data from v2:

```
std::vector<int> v(v2);
std::vector<int> v = v2;
```

Version ≥ C++11

Move construction (from another vector only), which moves data from v2:

```
std::vector<int> v(std::move(v2));
std::vector<int> v = std::move(v2);
```

Iterator (range) copy-construction, which copies elements into v:

```
// from another vector
std::vector<int> v(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}

// from an array
int z[] = { 1, 2, 3, 4 };
std::vector<int> v(z, z + 3);                   // v becomes {1, 2, 3}

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(list1.begin(), list1.end()); // v becomes {1, 2, 3}
```

Version ≥ C++11

Iterator move-construction, using std::make_move_iterator, which moves elements into v:

```
// from another vector
std::vector<int> v(std::make_move_iterator(v2.begin()),
                   std::make_move_iterator(v2.end());

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(std::make_move_iterator(list1.begin()),
                   std::make_move_iterator(list1.end()));
```

With the help of the assign() member function, a std::vector can be reinitialized after its construction:

```
v.assign(4, 100);                      // v becomes {100, 100, 100, 100}

v.assign(v2.begin(), v2.begin() + 3);  // v becomes {v2[0], v2[1], v2[2]}

int z[] = { 1, 2, 3, 4 };
v.assign(z + 1, z + 4);                // v becomes {2, 3, 4}
```

Section 49.3: Deleting Elements

Deleting the last element:

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back();                           // v becomes {1, 2}
```

Deleting all elements:

```
std::vector<int> v{ 1, 2, 3 };
v.clear();                              // v becomes an empty vector
```

Deleting element by index:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3);                 // v becomes {1, 2, 3, 5, 6}
```

Note: For a `vector` deleting an element which is not the last element, all elements beyond the deleted element have to be copied or moved to fill the gap, see the note below and std::list.

Deleting all elements in a range:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5);  // v becomes {1, 6}
```

Note: The above methods do not change the capacity of the vector, only the size. See Vector Size and Capacity.

The `erase` method, which removes a range of elements, is often used as a part of the **erase-remove** idiom. That is, first `std::remove` moves some elements to the end of the vector, and then `erase` chops them off. This is a relatively inefficient operation for any indices less than the last index of the vector because all elements after the erased segments must be relocated to new positions. For speed critical applications that require efficient removal of arbitrary elements in a container, see std::list.

Deleting elements by value:

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v becomes {1, 1, 3, 3}
```

Deleting elements by condition:

```
// std::remove_if needs a function, that takes a vector element as argument and returns true,
// if the element shall be removed
bool _predicate(const int& element) {
    return (element > 3); // This will cause all elements to be deleted that are larger than 3
}
...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v becomes {1, 2, 3}
```

Deleting elements by lambda, without creating additional predicate function

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
     [](auto& element){return element > 3;} ), v.end()
);
```

Deleting elements by condition from a loop:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // after erasing, 'it' will be set to the next element in v
    else
        ++it;             // manually set 'it' to the next element in v
}
```

While it is important *not* to increment `it` in case of a deletion, you should consider using a different method when then erasing repeatedly in a loop. Consider `remove_if` for a more efficient way.

Deleting elements by condition from a reverse loop:

```
std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
rev_itr it = v.rbegin();

while (it != v.rend()) { // after the loop only '0' will be in v
    int value = *it;
    if (value) {
        ++it;
        // See explanation below for the following line.
        it = rev_itr(v.erase(it.base()));
    } else
        ++it;
}
```

Note some points for the preceding loop:

- Given a reverse iterator `it` pointing to some element, the method base gives the regular (non-reverse) iterator pointing to the same element.
- `vector::erase(iterator)` erases the element pointed to by an iterator, and returns an iterator to the element that followed the given element.
- `reverse_iterator::reverse_iterator(iterator)` constructs a reverse iterator from an iterator.

Put altogether, the line `it = rev_itr(v.erase(it.base()))` says: take the reverse iterator `it`, have `v` erase the element pointed by its regular iterator; take the resulting iterator, construct a reverse iterator from it, and assign it to the reverse iterator `it`.

Deleting all elements using `v.clear()` does not free up memory (`capacity()` of the vector remains unchanged). To reclaim space, use:

```
std::vector<int>().swap(v);
```

Version ≥ C++11

`shrink_to_fit()` frees up unused vector capacity:

```
v.shrink_to_fit();
```

The `shrink_to_fit` does not guarantee to really reclaim space, but most current implementations do.

Section 49.4: Iterating Over std::vector

You can iterate over a `std::vector` in several ways. For each of the following sections, `v` is defined as follows:

```
std::vector<int> v;
```

Iterating in the Forward Direction

Version ≥ C++11

```
// Range based for
for(const auto& value: v) {
    std::cout << value << "\n";
}

// Using a for loop with iterator
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}

// Using for_each algorithm, using a function or functor:
void fun(int const& value) {
    std::cout << value << "\n";
}

std::for_each(std::begin(v), std::end(v), fun);
```

```
// Using for_each algorithm. Using a lambda:
std::for_each(std::begin(v), std::end(v), [](int const& value) {
    std::cout << value << "\n";
});
```

Version < C++11

```
// Using a for loop with iterator
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << "\n";
}
```

Iterating in the Reverse Direction

Version ≥ C++14

```
// There is no standard way to use range based for for this.
// See below for alternatives.

// Using for_each algorithm
// Note: Using a lambda for clarity. But a function or functor will work
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
    std::cout << value << "\n";
});

// Using a for loop with iterator
for(auto rit = std::rbegin(v); rit != std::rend(v); ++rit) {
    std::cout << *rit << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[v.size() - 1 - i] << "\n";
}
```

Though there is no built-in way to use the range based for to reverse iterate; it is relatively simple to fix this. The range based for uses `begin()` and `end()` to get iterators and thus simulating this with a wrapper object can achieve the results we require.

Version ≥ C++14

```
template<class C>
struct ReverseRange {
  C c; // could be a reference or a copy, if the original was a temporary
  ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
  ReverseRange(ReverseRange&&)=default;
  ReverseRange& operator=(ReverseRange&&)=delete;
  auto begin() const {return std::rbegin(c);}
  auto end()   const {return std::rend(c);}
};
// C is meant to be deduced, and perfect forwarded into
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
    std::vector<int> v { 1,2,3,4};
    for(auto const& value: make_ReverseRange(v)) {
        std::cout << value << "\n";
    }
}
```

Enforcing const elements

Since C++11 the `cbegin()` and `cend()` methods allow you to obtain a *constant iterator* for a vector, even if the vector is non-const. A constant iterator allows you to read but not modify the contents of the vector which is useful to enforce const correctness:

Version ≥ C++11

```
// forward iteration
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// reverse iteration
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// expects Functor::operand()(T&)
for_each(v.begin(), v.end(), Functor());

// expects Functor::operand()(const T&)
for_each(v.cbegin(), v.cend(), Functor())
```

Version ≥ C++17

as_const extends this to range iteration:

```
for (auto const& e : std::as_const(v)) {
  std::cout << e << '\n';
}
```

This is easy to implement in earlier versions of C++:

Version ≥ C++14

```
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
  return t;
}
```

A Note on Efficiency

Since the class `std::vector` is basically a class that manages a dynamically allocated contiguous array, the same principle explained here applies to C++ vectors. Accessing the vector's content by index is much more efficient when following the row-major order principle. Of course, each access to the vector also puts its management content into the cache as well, but as has been debated many times (notably here and here), the difference in performance for iterating over a `std::vector` compared to a raw array is negligible. So the same principle of efficiency for raw arrays in C also applies for C++'s `std::vector`.

Section 49.5: vector<bool>: The Exception To So Many, So Many Rules

The standard (section 23.3.7) specifies that a specialization of `vector<bool>` is provided, which optimizes space by packing the `bool` values, so that each takes up only one bit. Since bits aren't addressable in C++, this means that several requirements on `vector` are not placed on `vector<bool>`:

- The data stored is not required to be contiguous, so a `vector<bool>` can't be passed to a C API which expects a `bool` array.
- `at()`, `operator []`, and dereferencing of iterators do not return a reference to `bool`. Rather they return a

proxy object that (imperfectly) simulates a reference to a `bool` by overloading its assignment operators. As an example, the following code may not be valid for `std::vector<bool>`, because dereferencing an iterator does not return a reference:

Version ≥ C++11

```
std::vector<bool> v = {true, false};
for (auto &b: v) { } // error
```

Similarly, functions expecting a `bool&` argument cannot be used with the result of `operator []` or `at()` applied to `vector<bool>`, or with the result of dereferencing its iterator:

```
  void f(bool& b);
  f(v[0]);             // error
  f(*v.begin());       // error
```

The implementation of `std::vector<bool>` is dependent on both the compiler and architecture. The specialisation is implemented by packing `n` Booleans into the lowest addressable section of memory. Here, `n` is the size in bits of the lowest addressable memory. In most modern systems this is 1 byte or 8 bits. This means that one byte can store 8 Boolean values. This is an improvement over the traditional implementation where 1 Boolean value is stored in 1 byte of memory.

Note: The below example shows possible bitwise values of individual bytes in a traditional vs. optimized `vector<bool>`. This will not always hold true in all architectures. It is, however, a good way of visualising the optimization. In the below examples a byte is represented as [x, x, x, x, x, x, x, x].

Traditional *`std::vector<char>` storing 8 Boolean values:*

Version ≥ C++11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

Bitwise representation:

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

Specialized *`std::vector<bool>` storing 8 Boolean values:*

Version ≥ C++11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

Bitwise representation:

```
[1,0,0,0,1,0,1,1]
```

Notice in the above example, that in the traditional version of `std::vector<bool>`, 8 Boolean values take up 8 bytes of memory, whereas in the optimized version of `std::vector<bool>`, they only use 1 byte of memory. This is a significant improvement on memory usage. If you need to pass a `vector<bool>` to an C-style API, you may need to copy the values to an array, or find a better way to use the API, if memory and performance are at risk.

Section 49.6: Inserting Elements

Appending an element at the end of a vector (by copying/moving):

```
struct Point {
  double x, y;
  Point(double x, double y) : x(x), y(y) {}
};
std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p);  // p is copied into the vector.
```

Version ≥ C++11

Appending an element at the end of a vector by constructing the element in place:

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // The arguments are passed to the constructor of the
                           // given type (here Point). The object is constructed
                           // in the vector, avoiding a copy.
```

Note that `std::vector` does *not* have a `push_front()` member function due to performance reasons. Adding an element at the beginning causes all existing elements in the vector to be moved. If you want to frequently insert elements at the beginning of your container, then you might want to use `std::list` or `std::deque` instead.

Inserting an element at any position of a vector:

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9);          // v now contains {9, 1, 2, 3}
```

Version ≥ C++11

Inserting an element at any position of a vector by constructing the element in place:

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin()+1, 9);     // v now contains {1, 9, 2, 3}
```

Inserting another vector at any position of the vector:

```
std::vector<int> v(4);      // contains: 0, 0, 0, 0
std::vector<int> v2(2, 10); // contains: 10, 10
v.insert(v.begin()+2, v2.begin(), v2.end()); // contains: 0, 0, 10, 10, 0, 0
```

Inserting an array at any position of a vector:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
int a [] = {1, 2, 3}; // contains: 1, 2, 3
v.insert(v.begin()+1, a, a+sizeof(a)/sizeof(a[0])); // contains: 0, 1, 2, 3, 0, 0, 0
```

Use `reserve()` before inserting multiple elements if resulting vector size is known beforehand to avoid multiple reallocations (see vector size and capacity):

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

Be sure to not make the mistake of calling `resize()` in this case, or you will inadvertently create a vector with 200 elements where only the latter one hundred will have the value you intended.

Section 49.7: Using std::vector as a C array

There are several ways to use a `std::vector` as a C array (for example, for compatibility with C libraries). This is possible because the elements in a vector are stored contiguously.

Version ≥ C++11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

In contrast to solutions based on previous C++ standards (see below), the member function `.data()` may also be applied to empty vectors, because it doesn't cause undefined behavior in this case.

Before C++11, you would take the address of the vector's first element to get an equivalent pointer, if the vector isn't empty, these both methods are interchangeable:

```
int* p = &v[0];      // combine subscript operator and 0 literal

int* p = &v.front(); // explicitly reference the first element
```

Note: If the vector is empty, `v[0]` and `v.front()` are undefined and cannot be used.

When storing the base address of the vector's data, note that many operations (such as `push_back`, `resize`, etc.) can change the data memory location of the vector, thus invalidating previous data pointers. For example:

```
std::vector<int> v;
int* p = v.data();
v.resize(42);      // internal memory location changed; value of p is now invalid
```

Section 49.8: Finding an Element in std::vector

The function `std::find`, defined in the **`<algorithm>`** header, can be used to find an element in a `std::vector`.

`std::find` uses the `operator==` to compare elements for equality. It returns an iterator to the first element in the range that compares equal to the value.

If the element in question is not found, `std::find` returns `std::vector::end` (or `std::vector::cend` if the vector is `const`).

Version < C++11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

Version ≥ C++11

```
std::vector<int> v { 5, 4, 3, 2, 1 };

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

If you need to perform many searches in a large vector, then you may want to consider sorting the vector first, before using the binary_search algorithm.

To find the first element in a vector that satisfies a condition, std::find_if can be used. In addition to the two parameters given to std::find, std::find_if accepts a third argument which is a function object or function pointer to a predicate function. The predicate should accept an element from the container as an argument and return a value convertible to bool, without modifying the container:

Version < C++11

```
bool isEven(int val) {
    return (val % 2 == 0);
}

struct moreThan {
    moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};

static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` points to 8, the first even element

std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` is v.end(), as no element is greater than 10
```

Version ≥ C++11

```
// find the first value that is even
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [](int val){return val % 2 == 0;});
// `it` points to 8, the first even element

auto missing = std::find_if(v.begin(), v.end(), [](int val){return val > 10;});
// `missing` is v.end(), as no element is greater than 10
```

Section 49.9: Concatenating Vectors

One std::vector can be append to another by using the member function insert():

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

However, this solution fails if you try to append a vector to itself, because the standard specifies that iterators given to insert() must not be from the same range as the receiver object's elements.

Version ≥ c++11

Instead of using the vector's member functions, the functions std::begin() and std::end() can be used:

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

This is a more general solution, for example, because b can also be an array. However, also this solution doesn't allow you to append a vector to itself.

If the order of the elements in the receiving vector doesn't matter, considering the number of elements in each vector can avoid unnecessary copy operations:

```
if (b.size() < a.size())
  a.insert(a.end(), b.begin(), b.end());
else
  b.insert(b.end(), a.begin(), a.end());
```

Section 49.10: Matrices Using Vectors

Vectors can be used as a 2D matrix by defining them as a vector of vectors.

A matrix with 3 rows and 4 columns with each cell initialised as 0 can be defined as:

```
std::vector<std::vector<int> > matrix(3, std::vector<int>(4));
```

Version ≥ C++11

The syntax for initializing them using initialiser lists or otherwise are similar to that of a normal vector.

```
  std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                           {4,5,6,7},
                                           {8,9,10,11}
                                         };
```

Values in such a vector can be accessed similar to a 2D array

```
int var = matrix[0][2];
```

Iterating over the entire matrix is similar to that of a normal vector but with an extra dimension.

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
```

Version ≥ C++11

```
for(auto& row: matrix)
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

A vector of vectors is a convenient way to represent a matrix but it's not the most efficient: individual vectors are scattered around memory and the data structure isn't cache friendly.

Also, in a proper matrix, the length of every row must be the same (this isn't the case for a vector of vectors). The additional flexibility can be a source of errors.

Section 49.11: Using a Sorted Vector for Fast Element Lookup

The `<algorithm>` header provides a number of useful functions for working with sorted vectors.

An important prerequisite for working with sorted vectors is that the stored values are comparable with `<`.

An unsorted vector can be sorted by using the function `std::sort()`:

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

Sorted vectors allow efficient element lookup using the function `std::lower_bound()`. Unlike `std::find()`, this performs an efficient binary search on the vector. The downside is that it only gives valid results for sorted input ranges:

```
// search the vector for the first element with value 42
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // we found the element!
}
```

Note: If the requested value is not part of the vector, `std::lower_bound()` will return an iterator to the first element that is *greater* than the requested value. This behavior allows us to insert a new element at its right place in an already sorted vector:

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

If you need to insert a lot of elements at once, it might be more efficient to call `push_back()` for all them first and then call `std::sort()` once all elements have been inserted. In this case, the increased cost of the sorting can pay off against the reduced cost of inserting new elements at the end of the vector and not in the middle.

If your vector contains multiple elements of the same value, `std::lower_bound()` will try to return an iterator to the first element of the searched value. However, if you need to insert a new element *after* the last element of the searched value, you should use the function `std::upper_bound()` as this will cause less shifting around of elements:

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

If you need both the upper bound and the lower bound iterators, you can use the function `std::equal_range()` to retrieve both of them efficiently with one call:

```
std::pair<std::vector<int>::iterator,
          std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);
std::vector<int>::iterator lower_bound = rg.first;
std::vector<int>::iterator upper_bound = rg.second;
```

In order to test whether an element exists in a sorted vector (although not specific to vectors), you can use the function `std::binary_search()`:

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

Section 49.12: Reducing the Capacity of a Vector

A `std::vector` automatically increases its capacity upon insertion as needed, but it never reduces its capacity after element removal.

```
// Initialize a vector with 100 elements
std::vector<int> v(100);

// The vector's capacity is always at least as large as its size
auto const old_capacity = v.capacity();
// old_capacity >= 100

// Remove half of the elements
v.erase(v.begin() + 50, v.end());  // Reduces the size from 100 to 50 (v.size() == 50),
                                   // but not the capacity (v.capacity() == old_capacity)
```

To reduce its capacity, we can copy the contents of a vector to a new temporary vector. The new vector will have the minimum capacity that is needed to store all elements of the original vector. If the size reduction of the original vector was significant, then the capacity reduction for the new vector is likely to be significant. We can then swap the original vector with the temporary one to retain its minimized capacity:

```
std::vector<int>(v).swap(v);
```

Version ≥ C++11

In C++11 we can use the `shrink_to_fit()` member function for a similar effect:

```
v.shrink_to_fit();
```

Note: The `shrink_to_fit()` member function is a request and doesn't guarantee to reduce capacity.

Section 49.13: Vector size and capacity

Vector size is simply the number of elements in the vector:

1. Current vector **size** is queried by `size()` member function. Convenience `empty()` function returns `true` if size is 0:

    ```
    vector<int> v = { 1, 2, 3 }; // size is 3
    const vector<int>::size_type size = v.size();
    cout << size << endl; // prints 3
    cout << boolalpha << v.empty() << endl; // prints false
    ```

2. Default constructed vector starts with a size of 0:

    ```
    vector<int> v; // size is 0
    cout << v.size() << endl; // prints 0
    ```

3. Adding `N` elements to vector increases **size** by `N` (e.g. by `push_back()`, `insert()` or `resize()` functions).
4. Removing `N` elements from vector decreases **size** by `N` (e.g. by `pop_back()`, `erase()` or `clear()` functions).
5. Vector has an implementation-specific upper limit on its size, but you are likely to run out of RAM before reaching it:

    ```
    vector<int> v;
    ```

```
const vector<int>::size_type max_size = v.max_size();
cout << max_size << endl; // prints some large number
v.resize( max_size ); // probably won't work
v.push_back( 1 ); // definitely won't work
```

Common mistake: **size** is not necessarily (or even usually) `int`:

```
// !!!bad!!!evil!!!
vector<int> v_bad( N, 1 ); // constructs large N size vector
for( int i = 0; i < v_bad.size(); ++i ) { // size is not supposed to be int!
    do_something( v_bad[i] );
}
```

Vector capacity differs from **size**. While **size** is simply how many elements the vector currently has, **capacity** is for how many elements it allocated/reserved memory for. That is useful, because too frequent (re)allocations of too large sizes can be expensive.

1. Current vector **capacity** is queried by `capacity()` member function. **Capacity** is always greater or equal to **size**:

   ```
   vector<int> v = { 1, 2, 3 }; // size is 3, capacity is >= 3
   const vector<int>::size_type capacity = v.capacity();
   cout << capacity << endl; // prints number >= 3
   ```

2. You can manually reserve capacity by `reserve( N )` function (it changes vector capacity to `N`):

   ```
   // !!!bad!!!evil!!!
   vector<int> v_bad;
   for( int i = 0; i < 10000; ++i ) {
       v_bad.push_back( i ); // possibly lot of reallocations
   }

   // good
   vector<int> v_good;
   v_good.reserve( 10000 ); // good! only one allocation
   for( int i = 0; i < 10000; ++i ) {
       v_good.push_back( i ); // no allocations needed anymore
   }
   ```

3. You can request for the excess capacity to be released by `shrink_to_fit()` (but the implementation doesn't have to obey you). This is useful to conserve used memory:

   ```
   vector<int> v = { 1, 2, 3, 4, 5 }; // size is 5, assume capacity is 6
   v.shrink_to_fit(); // capacity is 5 (or possibly still 6)
   cout << boolalpha << v.capacity() == v.size() << endl; // prints likely true (but possibly false)
   ```

Vector partly manages capacity automatically, when you add elements it may decide to grow. Implementers like to use 2 or 1.5 for the grow factor (golden ratio would be the ideal value - but is impractical due to being rational number). On the other hand vector usually do not automatically shrink. For example:

```
vector<int> v; // capacity is possibly (but not guaranteed) to be 0
v.push_back( 1 ); // capacity is some starter value, likely 1
v.clear(); // size is 0 but capacity is still same as before!
```