

# 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::istreambuf` 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 internal encoding.

More on streams: `<ostream>` 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 and stream insertion operator `<<`.

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

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

```
std::ofstream of("foo.txt");
if(of.is_open())
    char data[] = "foo";
    of.write(data, 3);
}
```

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

```
of << "Hello World!"; // this operation might fail for any reason
if (of.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 if("foo.txt"); // ifstream: opens file "foo.txt" for reading only.
std::ofstream of("foo.txt"); // ofstream: opens file "foo.txt" for writing only.
```

C++ Notes for Professionals

### 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_r`).
- 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, `constexpr` or literal strings, to tokenize any of those with `std::strtok` or to operate on a `std::string` whose 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.c_str(), " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

#### Like 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, also inline tokenization thereby supporting the generation of a `const` `vector<string>`, and by adding support multiple delimiting white-space character.

```
// String to tokenize
const std::string str("The quick brown fox");
std::istringstream istr(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(istr);
std::istream_iterator<string> end(istr);
```

#### Like 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

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

// 1) Using aggregate initialization

```
std::array<int, 2> a{ 0, 1, 2 };
// or equivalently
std::array<int, 2> a = { 0, 1, 2 };
```

// 2) Using the copy constructor

```
std::array<int, 2> a1{ 0, 1, 2 };
std::array<int, 2> a2(a1);
// or equivalently
std::array<int, 2> a2 = a1;
```

// 3) Using the move constructor

```
std::array<int, 2> a = std::array<int, 2>{ 0, 1, 2 };
```

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 notation

```
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 } };
```

// 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 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

// 3) Using the copy constructor

```
std::array<A, 2> a1{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
std::array<A, 2> a2(a1);
// or equivalently
std::array<A, 2> a2 = a1;
```

// 4) Using the move constructor

```
std::array<A, 2> a = std::array<A, 2>{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
```

C++ Notes for Professionals

305

600+ pages  
of professional hints and tricks

# Contents

<b>About</b>	1
<b>Chapter 1: Getting started with C++</b>	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
<b>Chapter 2: Literals</b>	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
<b>Chapter 3: operator precedence</b>	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
<b>Chapter 4: Floating Point Arithmetic</b>	17
Section 4.1: Floating Point Numbers are Weird	17
<b>Chapter 5: Bit Operators</b>	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
<b>Chapter 6: Bit Manipulation</b>	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
<b>Chapter 7: Bit fields</b>	27
Section 7.1: Declaration and Usage	27
<b>Chapter 8: Arrays</b>	28
Section 8.1: Array initialization	28
Section 8.2: A fixed size row array matrix (that is, a 2D row array)	29
Section 8.3: Dynamically sized row 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
<b>Chapter 9: Iterators</b> .....	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
<b>Chapter 10: Basic input/output in c++</b> .....	43
Section 10.1: user input and standard output .....	43
<b>Chapter 11: Loops</b> .....	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
<b>Chapter 12: File I/O</b> .....	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
<b>Chapter 13: C++ Streams</b> .....	65
Section 13.1: String streams .....	65
Section 13.2: Printing collections with ostream .....	66
<b>Chapter 14: Stream manipulators</b> .....	68
Section 14.1: Stream manipulators .....	68
Section 14.2: Output stream manipulators .....	73
Section 14.3: Input stream manipulators .....	75
<b>Chapter 15: Flow Control</b> .....	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
<b>Chapter 16: Metaprogramming .....</b>	<b>86</b>
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
<b>Chapter 17: const keyword .....</b>	<b>95</b>
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
<b>Chapter 18: mutable keyword .....</b>	<b>99</b>
Section 18.1: mutable lambdas .....	99
Section 18.2: non-static class member modifier .....	99
<b>Chapter 19: Friend keyword .....</b>	<b>101</b>
Section 19.1: Friend function .....	101
Section 19.2: Friend method .....	102
Section 19.3: Friend class .....	102
<b>Chapter 20: Type Keywords .....</b>	<b>104</b>
Section 20.1: class .....	104
Section 20.2: enum .....	105
Section 20.3: struct .....	106
Section 20.4: union .....	106
<b>Chapter 21: Basic Type Keywords .....</b>	<b>108</b>
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
<b>Chapter 22: Variable Declaration Keywords .....</b>	<b>111</b>
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
<b>Chapter 23: Keywords .....</b>	<b>114</b>

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
<b>Chapter 24: Returning several values from a function .....</b>	<b>122</b>
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
<b>Chapter 25: Polymorphism .....</b>	<b>129</b>
Section 25.1: Define polymorphic classes .....	129
Section 25.2: Safe downcasting .....	130
Section 25.3: Polymorphism & Destructors .....	131
<b>Chapter 26: References .....</b>	<b>133</b>
Section 26.1: Defining a reference .....	133
<b>Chapter 27: Value and Reference Semantics .....</b>	<b>134</b>
Section 27.1: Definitions .....	134
Section 27.2: Deep copying and move support .....	134
<b>Chapter 28: C++ function "call by value" vs. "call by reference" .....</b>	<b>138</b>
Section 28.1: Call by value .....	138
<b>Chapter 29: Copying vs Assignment .....</b>	<b>140</b>
Section 29.1: Assignment Operator .....	140
Section 29.2: Copy Constructor .....	140
Section 29.3: Copy Constructor Vs Assignment Constructor .....	141
<b>Chapter 30: Pointers .....</b>	<b>143</b>
Section 30.1: Pointer Operations .....	143
Section 30.2: Pointer basics .....	143
Section 30.3: Pointer Arithmetic .....	145
<b>Chapter 31: Pointers to members .....</b>	<b>147</b>
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
<b>Chapter 32: The This Pointer .....</b>	<b>150</b>
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
<b>Chapter 33: Smart Pointers .....</b>	<b>158</b>
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
<b>Chapter 34: Classes/Structures</b>	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
<b>Chapter 35: Function Overloading</b>	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
<b>Chapter 36: Operator Overloading</b>	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
<b>Chapter 37: Function Template Overloading</b>	213
Section 37.1: What is a valid function template overloading?	213
<b>Chapter 38: Virtual Member Functions</b>	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
<b>Chapter 39: Inline functions</b>	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

<b>Chapter 40: Special Member Functions</b>	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
<b>Chapter 41: Non-Static Member Functions</b>	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
<b>Chapter 42: Constant class member functions</b>	235
Section 42.1: constant member function	235
<b>Chapter 43: C++ Containers</b>	236
Section 43.1: C++ Containers Flowchart	236
<b>Chapter 44: Namespaces</b>	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
<b>Chapter 45: Header Files</b>	246
Section 45.1: Basic Example	246
Section 45.2: Templates in Header Files	247
<b>Chapter 46: Using declaration</b>	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
<b>Chapter 47: std::string</b>	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
<b>Chapter 48: std::array</b> .....	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
<b>Chapter 49: std::vector</b> .....	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
<b>Chapter 50: std::map</b> .....	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
<b>Chapter 51: std::optional</b> .....	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
<b>Chapter 52: std::function: To wrap any element that is callable</b> .....	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
<b>Chapter 53: std::forward_list</b> .....	305



Section 53.1: Example .....	305
Section 53.2: Methods .....	305
<b>Chapter 54: std::pair</b> .....	307
Section 54.1: Compare operators .....	307
Section 54.2: Creating a Pair and accessing the elements .....	307
<b>Chapter 55: std::atomics</b> .....	309
Section 55.1: atomic types .....	309
<b>Chapter 56: std::variant</b> .....	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
<b>Chapter 57: std::io manip</b> .....	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
<b>Chapter 58: std::any</b> .....	317
Section 58.1: Basic usage .....	317
<b>Chapter 59: std::set and std::multiset</b> .....	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
<b>Chapter 60: std::integer_sequence</b> .....	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
<b>Chapter 61: Using std::unordered_map</b> .....	328
Section 61.1: Declaration and Usage .....	328
Section 61.2: Some Basic Functions .....	328
<b>Chapter 62: Standard Library Algorithms</b> .....	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
<b>Chapter 63: The ISO C++ Standard</b> .....	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

<b>Chapter 64: Inline variables</b>	344
Section 64.1: Defining a static data member in the class definition	344
<b>Chapter 65: Random number generation</b>	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
<b>Chapter 66: Date and time using &lt;chrono&gt; header</b>	347
Section 66.1: Measuring time using <chrono>	347
Section 66.2: Find number of days between two dates	347
<b>Chapter 67: Sorting</b>	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 specified ordering	354
<b>Chapter 68: Enumeration</b>	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
<b>Chapter 69: Iteration</b>	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
<b>Chapter 70: Regular expressions</b>	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
<b>Chapter 71: Implementation-defined behavior</b>	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
<b>Chapter 72: Exceptions</b>	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
<b>Chapter 73: Lambdas</b> .....	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
<b>Chapter 74: Value Categories</b> .....	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
<b>Chapter 75: Preprocessor</b> .....	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
<b>Chapter 76: Data Structures in C++</b> .....	410
Section 76.1: Linked List implementation in C++ .....	410
<b>Chapter 77: Templates</b> .....	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
<b>Chapter 78: Expression templates .....</b>	<b>425</b>
Section 78.1: A basic example illustrating expression templates .....	425
<b>Chapter 79: Curiously Recurring Template Pattern (CRTP) .....</b>	<b>429</b>
Section 79.1: The Curiously Recurring Template Pattern (CRTP) .....	429
Section 79.2: CRTP to avoid code duplication .....	430
<b>Chapter 80: Threading .....</b>	<b>432</b>
Section 80.1: Creating a <code>std::thread</code> .....	432
Section 80.2: Passing a reference to a thread .....	434
Section 80.3: Using <code>std::async</code> instead of <code>std::thread</code> .....	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
<b>Chapter 81: Thread synchronization structures .....</b>	<b>443</b>
Section 81.1: <code>std::condition_variable</code> , <code>any</code> , <code>std::cv_status</code> .....	443
Section 81.2: <code>std::shared_lock</code> .....	443
Section 81.3: <code>std::call_once</code> , <code>std::once_flag</code> .....	443
Section 81.4: Object locking for efficient access .....	444
<b>Chapter 82: The Rule of Three, Five, And Zero .....</b>	<b>446</b>
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
<b>Chapter 83: RAI: Resource Acquisition Is Initialization .....</b>	<b>451</b>
Section 83.1: Locking .....	451
Section 83.2: <code>ScopeSuccess</code> (C++17) .....	452
Section 83.3: <code>ScopeFail</code> (C++17) .....	453
Section 83.4: <code>Finally/ScopeExit</code> .....	454
<b>Chapter 84: RTTI: Run-Time Type Information .....</b>	<b>455</b>
Section 84.1: <code>dynamic_cast</code> .....	455
Section 84.2: The <code>typeid</code> keyword .....	455
Section 84.3: Name of a type .....	456
Section 84.4: When to use which cast in C++ .....	456
<b>Chapter 85: Mutexes .....</b>	<b>457</b>
Section 85.1: Mutex Types .....	457
Section 85.2: <code>std::lock</code> .....	457
Section 85.3: <code>std::unique_lock</code> , <code>std::shared_lock</code> , <code>std::lock_guard</code> .....	457
Section 85.4: Strategies for lock classes: <code>std::try_to_lock</code> , <code>std::adopt_lock</code> , <code>std::defer_lock</code> .....	458
Section 85.5: <code>std::mutex</code> .....	459
Section 85.6: <code>std::scoped_lock</code> (C++ 17) .....	459
<b>Chapter 86: Recursive Mutex .....</b>	<b>460</b>
Section 86.1: <code>std::recursive_mutex</code> .....	460
<b>Chapter 87: Semaphore .....</b>	<b>461</b>

Section 87.1: Semaphore C++ 11 .....	461
Section 87.2: Semaphore class in action .....	461
<b>Chapter 88: Futures and Promises</b> .....	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
<b>Chapter 89: Atomic Types</b> .....	468
Section 89.1: Multi-threaded Access .....	468
<b>Chapter 90: Type Erasure</b> .....	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
<b>Chapter 91: Explicit type conversions</b> .....	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
<b>Chapter 92: Unnamed types</b> .....	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
<b>Chapter 93: Type Traits</b> .....	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
<b>Chapter 94: Return Type Covariance</b> .....	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
<b>Chapter 95: Layout of object types</b> .....	497
Section 95.1: Class types .....	497
Section 95.2: Arithmetic types .....	499
Section 95.3: Arrays .....	500
<b>Chapter 96: Type Inference</b> .....	501
Section 96.1: Data Type: Auto .....	501
Section 96.2: Lambda auto .....	501
Section 96.3: Loops and auto .....	501

<b>Chapter 97: Typedef and type aliases</b>	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
<b>Chapter 98: type deduction</b>	505
Section 98.1: Template parameter deduction for constructors	505
Section 98.2: Auto Type Deduction	505
Section 98.3: Template Type Deduction	506
<b>Chapter 99: Trailing return type</b>	508
Section 99.1: Avoid qualifying a nested type name	508
Section 99.2: Lambda expressions	508
<b>Chapter 100: Alignment</b>	509
Section 100.1: Controlling alignment	509
Section 100.2: Querying the alignment of a type	509
<b>Chapter 101: Perfect Forwarding</b>	511
Section 101.1: Factory functions	511
<b>Chapter 102: decltype</b>	512
Section 102.1: Basic Example	512
Section 102.2: Another example	512
<b>Chapter 103: SFINAE (Substitution Failure Is Not An Error)</b>	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
<b>Chapter 104: Undefined Behavior</b>	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
<b>Chapter 105: Overload resolution</b> .....	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
<b>Chapter 106: Move Semantics</b> .....	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
<b>Chapter 107: Pimpl Idiom</b> .....	549
Section 107.1: Basic Pimpl idiom .....	549
<b>Chapter 108: auto</b> .....	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
<b>Chapter 109: Copy Elision</b> .....	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
<b>Chapter 110: Fold Expressions</b> .....	559
Section 110.1: Unary Folds .....	559
Section 110.2: Binary Folds .....	559
Section 110.3: Folding over a comma .....	560
<b>Chapter 111: Unions</b> .....	561
Section 111.1: Undefined Behavior .....	561
Section 111.2: Basic Union Features .....	561
Section 111.3: Typical Use .....	561
<b>Chapter 112: Design pattern implementation in C++</b> .....	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
<b>Chapter 113: Singleton Design Pattern .....</b>	<b>572</b>
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
<b>Chapter 114: User-Defined Literals .....</b>	<b>575</b>
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
<b>Chapter 115: Memory management .....</b>	<b>578</b>
Section 115.1: Free Storage (Heap, Dynamic Allocation ...) .....	578
Section 115.2: Placement new .....	579
Section 115.3: Stack .....	580
<b>Chapter 116: C++11 Memory Model .....</b>	<b>581</b>
Section 116.1: Need for Memory Model .....	582
Section 116.2: Fence example .....	584
<b>Chapter 117: Scopes .....</b>	<b>585</b>
Section 117.1: Global variables .....	585
Section 117.2: Simple block scope .....	585
<b>Chapter 118: static_assert .....</b>	<b>587</b>
Section 118.1: static_assert .....	587
<b>Chapter 119: constexpr .....</b>	<b>588</b>
Section 119.1: constexpr variables .....	588
Section 119.2: Static if statement .....	589
Section 119.3: constexpr functions .....	590
<b>Chapter 120: One Definition Rule (ODR) .....</b>	<b>592</b>
Section 120.1: ODR violation via overload resolution .....	592
Section 120.2: Multiply defined function .....	592
Section 120.3: Inline functions .....	593
<b>Chapter 121: Unspecified behavior .....</b>	<b>595</b>
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
<b>Chapter 122: Argument Dependent Name Lookup .....</b>	<b>600</b>
Section 122.1: What functions are found .....	600
<b>Chapter 123: Attributes .....</b>	<b>601</b>
Section 123.1: [[fallthrough]] .....	601
Section 123.2: [[nodiscard]] .....	601
Section 123.3: [[deprecated]] and [[deprecated("reason")]] .....	602



Section 123.4: <code>[[maybe_unused]]</code> .....	602
Section 123.5: <code>[[noreturn]]</code> .....	603
<b>Chapter 124: Recursion in C++</b> .....	605
Section 124.1: Using tail recursion and Fibonacci-style recursion to solve the Fibonacci sequence .....	605
Section 124.2: Recursion with memoization .....	605
<b>Chapter 125: Arithmetic Metaprogramming</b> .....	607
Section 125.1: Calculating power in $O(\log n)$ .....	607
<b>Chapter 126: Callable Objects</b> .....	609
Section 126.1: Function Pointers .....	609
Section 126.2: Classes with <code>operator()</code> (Functors) .....	609
<b>Chapter 127: Client server examples</b> .....	611
Section 127.1: Hello TCP Client .....	611
Section 127.2: Hello TCP Server .....	612
<b>Chapter 128: Const Correctness</b> .....	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
<b>Chapter 129: Parameter packs</b> .....	624
Section 129.1: A template with a parameter pack .....	624
Section 129.2: Expansion of a parameter pack .....	624
<b>Chapter 130: Build Systems</b> .....	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
<b>Chapter 131: Concurrency With OpenMP</b> .....	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
<b>Chapter 132: Resource Management</b> .....	633
Section 132.1: Resource Acquisition Is Initialization .....	633
Section 132.2: Mutexes & Thread Safety .....	634
<b>Chapter 133: Storage class specifiers</b> .....	636
Section 133.1: <code>extern</code> .....	636
Section 133.2: <code>register</code> .....	637
Section 133.3: <code>static</code> .....	637
Section 133.4: <code>auto</code> .....	638
Section 133.5: <code>mutable</code> .....	638
<b>Chapter 134: Linkage specifications</b> .....	640
Section 134.1: Signal handler for Unix-like operating system .....	640
Section 134.2: Making a C library header compatible with C++ .....	640
<b>Chapter 135: Digit separators</b> .....	642
Section 135.1: Digit Separator .....	642
<b>Chapter 136: C incompatibilities</b> .....	643

Section 136.1: Reserved Keywords .....	643
Section 136.2: Weakly typed pointers .....	643
Section 136.3: goto or switch .....	643
<b>Chapter 137: Side by Side Comparisons of classic C++ examples solved via C++ vs C++11 vs C++14 vs C++17</b> .....	644
Section 137.1: Looping through a container .....	644
<b>Chapter 138: Compiling and Building</b> .....	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
<b>Chapter 139: Common compile/linker errors (GCC)</b> .....	661
Section 139.1: undefined reference to <code>`***'</code> .....	661
Section 139.2: error: <code>'***'</code> was not declared in this scope .....	661
Section 139.3: fatal error: <code>***</code> : No such file or directory .....	663
<b>Chapter 140: More undefined behaviors in C++</b> .....	664
Section 140.1: Referring to non-static members in initializer lists .....	664
<b>Chapter 141: Unit Testing in C++</b> .....	665
Section 141.1: Google Test .....	665
Section 141.2: Catch .....	665
<b>Chapter 142: C++ Debugging and Debug-prevention Tools &amp; Techniques</b> .....	667
Section 142.1: Static analysis .....	667
Section 142.2: Segfault analysis with GDB .....	668
Section 142.3: Clean code .....	669
<b>Chapter 143: Optimization in C++</b> .....	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
<b>Chapter 144: Optimization</b> .....	676
Section 144.1: Inline Expansion/Inlining .....	676
Section 144.2: Empty base optimization .....	676
<b>Chapter 145: Profiling</b> .....	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
<b>Chapter 146: Refactoring Techniques</b> .....	681
Section 146.1: Goto Cleanup .....	681
<b>Credits</b> .....	682
<b>You may also like</b> .....	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 C++ *Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

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](mailto: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** 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 tell 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  $\geq$  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 0.333333333333... in base-10.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a = 00111111101110011001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.1
double b = 00111111110010011001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.2
double c = 00111111110100110011001100110011001100110011001100110011001100110011; //imperfect
representation of 0.3
double a + b = 00111111110100110011001100110011001100110011001100110011001100110010; //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:  $a \oplus c = \sim(a \& c) \& (a | c)$

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 multiplication:

```
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 type `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  $2 \times n = 34$ . And more generally the sum of this sequence can't exceed  $2 \times n$ .

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;

```