

# Introduction to Scientific Programming in C++23/Fortran2018

The Art of HPC, volume 3

Victor Eijkhout

2017–2025, formatted September 2, 2025

Book and slides download: <https://tinyurl.com/vle322course>

Public repository: <https://bitbucket.org/VictorEijkhout/intro-programming-public>

This book is published under the CC-BY 4.0 license.



## Contents

	3.4.5	Functions 31
	3.4.6	Scope 31
	3.4.7	Classes 31
	3.4.8	Arrays vs vectors 31
	3.4.9	Strings 31
	3.4.10	Other remarks 32
	II C++ 33	
	4	<b>Basic elements of C++ 35</b>
	4.1	<i>From the ground up:</i>
		<i>Compiling C++ 35</i>
	4.1.1	A quick word about unix commands 37
	4.1.2	Build environments 37
	4.1.3	C++ is a moving target 38
	4.2	<i>Statements 38</i>
	4.2.1	Language vs library and about: using 40
	4.3	<i>Variables 40</i>
	4.3.1	Variable declarations 41
	4.3.2	Initialization 42
	4.3.3	Assignments 42
	4.3.4	Datatypes 43
	4.4	<i>Input/Output, or I/O as we say 46</i>
	4.5	<i>Expressions 47</i>
	4.5.1	Numerical expressions 47
	4.5.2	Truth values 48
	4.5.3	Type conversions 48
	4.5.4	Characters and strings 49
	4.6	<i>Advanced topics 50</i>
	4.6.1	The main program and the return statement 50
	4.6.2	Identifier names 51
	4.7	<i>C differences 51</i>
	4.7.1	Boolean 51
	4.8	<i>Review questions 51</i>
	5	<b>Conditionals 53</b>
	5.1	<i>Conditionals 53</i>
	5.2	<i>Operators 54</i>
	5.2.1	Bitwise logic 55
	5.3	<i>Switch statement 56</i>
	5.4	<i>Scopes 57</i>
	5.5	<i>Advanced topics 57</i>
	5.5.1	Short-circuit evaluation 57

5.5.2	Ternary if	58	8.3	<i>Scope and memory</i>	95
5.5.3	Initializer	58	8.4	<i>Review questions</i>	96
5.6	<i>Review questions</i>	59	9	<b>Classes and objects</b>	99
6	<b>Looping</b>	61	9.1	<i>What is an object?</i>	99
6.1	<i>The ‘for’ loop</i>	61	9.1.1	First example: points in the plane	99
6.1.1	Loop variable	62	9.1.2	Constructor	101
6.1.2	Stopping test	63	9.1.3	Data members	101
6.1.3	Increment	63	9.1.4	Methods	102
6.1.4	Loop body	65	9.1.5	Initialization	103
6.2	<i>Nested loops</i>	65	9.1.6	Methods, a deeper dive	105
6.3	<i>Looping until</i>	66	9.1.7	Default constructor	107
6.3.1	While loops	68	9.1.8	Data member access; invariants	109
6.4	<i>Advanced topics</i>	70	9.1.9	Examples	110
6.4.1	Parallelism	70	9.2	<i>Inclusion relations between classes</i>	111
6.5	<i>Exercises</i>	71	9.2.1	Literal and figurative has-a	112
7	<b>Functions</b>	73	9.3	<i>Inheritance</i>	114
7.1	<i>Function definition and call</i>	73	9.3.1	Methods of base and derived classes	115
7.1.1	Formal definition of a function definition	75	9.3.2	Virtual methods	115
7.1.2	Function call	76	9.3.3	Friend classes	117
7.1.3	Why use functions?	77	9.3.4	Multiple inheritance	117
7.2	<i>Anatomy of a function definition and call</i>	78	9.4	<i>More about constructors</i>	118
7.3	<i>Definition vs declaration</i>	78	9.4.1	Delegating constructors	118
7.4	<i>Void functions</i>	79	9.4.2	Copy constructor	119
7.5	<i>Parameter passing</i>	80	9.4.3	Destructor	120
7.5.1	Pass by value	80	9.5	<i>Advanced topics</i>	121
7.5.2	Pass by reference	82	9.5.1	Static variables and methods	121
7.6	<i>Recursive functions</i>	84	9.5.2	Inheriting from standard containers	123
7.7	<i>Other function topics</i>	87	9.5.3	Class declarations	124
7.7.1	Default arguments	87	9.5.4	Returning by reference	125
7.7.2	Polymorphic functions	87	9.5.5	Accessor functions	126
7.7.3	Math functions	87	9.5.6	Polymorphism	127
7.7.4	Trailing return type	88	9.5.7	Operator overloading	127
7.7.5	Stack overflow	88	9.5.8	Constructors and contained classes	129
7.8	<i>Review questions</i>	89	9.5.9	‘this’ pointer	129
8	<b>Scope</b>	91	9.5.10	Deducing this	131
8.1	<i>Scope rules</i>	91	9.5.11	Mutable data	131
8.1.1	Lexical scope	91			
8.1.2	Shadowing	91			
8.1.3	Lifetime versus reachability	92			
8.1.4	Scope subtleties	93			
8.2	<i>Static variables</i>	94			

9.5.12	Lazy evaluation	132	10.10	<i>C style arrays</i>	162
9.6	<i>Review questions</i>	133	10.10.1	Allocation	162
10	<b>Arrays</b>	135	10.10.2	Indexing and range-based loops	162
10.1	<i>Some simple examples</i>	135	10.10.3	C-style arrays and subprograms	163
10.1.1	Vector creation	135	10.10.4	Size of arrays	164
10.1.2	Initialization	136	10.10.5	Multi-dimensional arrays	165
10.1.3	Element access	137	10.10.6	Memory layout	165
10.1.4	Access out of bounds	137	10.11	<i>Exercises</i>	166
10.2	<i>Going over all vector elements</i>	138	11	<b>Strings</b>	169
10.2.1	Ranging over a vector	139	11.1	<i>Characters</i>	169
10.2.2	Ranging over the indices	140	11.2	<i>Basic string stuff</i>	169
10.2.3	Ranging by reference	141	11.3	<i>String streams</i>	173
10.2.4	Arrays and while loops	142	11.4	<i>Advanced topics</i>	173
10.3	<i>Vector are a class</i>	142	11.4.1	<i>String views</i>	173
10.3.1	Vector methods	142	11.4.2	<i>Raw string literals</i>	173
10.3.2	Vectors are dynamic	143	11.4.3	<i>String literal suffix</i>	174
10.4	<i>The Array class</i>	144	11.4.4	<i>Conversion to/from string</i>	174
10.4.1	Initialization	144	11.4.5	<i>Unicode</i>	175
10.4.2	Pass as argument	145	11.5	<i>C strings</i>	175
10.5	<i>Vectors and functions</i>	145	12	<b>Input/output</b>	177
10.5.1	Pass vector to function	145	12.1	<i>Streams vs the format library</i>	177
10.5.2	Vector as function return	147	12.2	<i>Using the format library</i>	178
10.6	<i>Vectors in classes</i>	148	12.2.1	<i>Numbered arguments</i>	178
10.6.1	Timing	149	12.2.2	<i>Align and padding</i>	179
10.7	<i>Wrapping a vector in an object</i>	150	12.2.3	<i>Number bases</i>	180
10.7.1	Public inheritance	151	12.2.4	<i>Floating point numbers</i>	180
10.8	<i>Multi-dimensional cases</i>	151	12.2.5	<i>Truth values</i>	181
10.8.1	Matrix as vector of vectors	151	12.2.6	<i>Limitations</i>	181
10.8.2	A better matrix class	153	12.3	<i>About fmtlib</i>	182
10.8.3	Span and mdspan	154	12.3.1	<i>Construct a string</i>	182
10.9	<i>Advanced topics</i>	156	12.3.2	<i>Output a range</i>	182
10.9.1	The size of a vector	156	12.3.3	<i>more</i>	182
10.9.2	Loop index type	156	12.4	<i>Stream-based formatting</i>	183
10.9.3	Container copying	157	12.4.1	<i>Screen output</i>	183
10.9.4	Failed allocation	157	12.4.2	<i>Floating point output</i>	186
10.9.5	Stack and heap allocation	157	12.4.3	<i>Boolean output</i>	188
10.9.6	Vector of bool	159	12.4.4	<i>Saving and restoring settings</i>	188
10.9.7	Span and mdspan	159	12.5	<i>File output</i>	188
10.9.8	Size and signedness	161			

12.5.1	Text output	188	14.2.3	Vector operations through iterators	215
12.5.2	Binary output	189	14.2.4	Forming sub-arrays	217
12.6	<i>Output your own classes</i>	190	14.3	<i>Algorithms on ranges</i>	218
12.6.1	Stream formatting	190	14.3.1	Test Any/all	218
12.7	<i>Output buffering</i>	191	14.3.2	Apply to each	220
12.7.1	The need for flushing	191	14.3.3	Iterator result	221
12.7.2	Performance considerations	191	14.3.4	Mapping	221
12.8	<i>Input</i>	192	14.3.5	Reduction	221
12.8.1	File input	193	14.3.6	Sorting	223
12.8.2	Input streams	194	14.4	<i>Parallel execution policies</i>	224
12.8.3	C-style file handling	194	14.5	<i>Classification of algorithms</i>	225
12.8.4	Align and padding	194	14.5.1	Non-parallel algorithms	225
12.8.5	Construct a string	195	14.5.2	Parallel algorithms	225
12.8.6	Number bases	195	14.6	<i>Advanced topics</i>	226
12.8.7	Output your own classes	195	14.6.1	Utilities	226
12.8.8	Output a range	196	14.6.2	Range types	226
13	<b>Lambda expressions</b>	197	14.6.3	Make your own iterator	227
13.1	<i>Lambda expressions as function argument</i>	198	15	<b>References</b>	229
13.1.1	Lambda members of classes	199	15.1	Reference	229
13.2	<i>Captures</i>	200	15.2	<i>Pass by reference</i>	229
13.2.1	Capture by reference	202	15.3	<i>Reference to class members</i>	230
13.2.2	Capturing ‘this’	203	15.4	<i>Reference to array members</i>	232
13.3	<i>More</i>	203	15.5	Addresses	233
13.3.1	Making lambda stateful	203	16	<b>Pointers</b>	235
13.3.2	Generic lambdas	204	16.1	<i>Pointer usage</i>	235
13.3.3	Algorithms	205	16.2	<i>How smart pointers prevent memory problems</i>	238
13.3.4	C-style function pointers	205	16.2.1	Memory leaks	238
14	<b>Iterators, Algorithms, Ranges</b>	207	16.2.2	Memory leaks and exceptions	239
14.1	<i>Ranges</i>	207	16.2.3	Use after free	240
14.1.1	Introduction	208	16.3	<i>Advanced topics</i>	240
14.1.2	Views	209	16.3.1	Get the pointed data	240
14.1.3	Example: sum of squares	212	16.3.2	Unique pointers	240
14.1.4	Infinite sequences	212	16.3.3	Base and derived pointers	241
14.2	<i>Iterators</i>	213	16.3.4	Shared pointer to ‘this’	241
14.2.1	Iterating with iterators	213	16.3.5	Weak pointers	242
14.2.2	Why still iterators, why not	215	16.3.6	Null pointer	242

16.3.7	Pointers to non-objects	242	19.4.3	Global variables and header files	272
16.4	<i>Smart pointers vs C pointers</i>	243	19.5	<b>Modules</b>	272
16.4.1	Smart pointers versus C-style address pointers	243	19.5.1	Program structure with modules	273
17	<b>C-style pointers and arrays</b>	245	19.5.2	Implementation and interface units	273
17.1	<i>What is a pointer</i>	245	19.5.3	<b>Partitions</b>	273
17.2	<i>Pointers and addresses, C style</i>	245	19.5.4	More	274
17.3	<i>Arrays and pointers</i>	248	20	<b>Namespaces</b>	275
17.4	<i>Pointer arithmetic</i>	250	20.1	<i>Solving name conflicts</i>	275
17.5	<i>Multi-dimensional arrays</i>	250	20.2	<i>Namespace header files</i>	276
17.6	<i>Parameter passing</i>	250	20.3	<i>Namespace versioning for libraries</i>	277
17.6.1	Allocation	251	20.4	<i>Best practices</i>	279
17.6.2	Use of <code>new</code>	253	21	<b>Preprocessor</b>	281
17.7	<i>Memory leaks</i>	254	21.1	<i>Include files</i>	281
17.8	<i>Const pointers</i>	254	21.1.1	Kinds of includes	282
18	<b>Const</b>	257	21.1.2	Search paths	282
18.1	<i>Const arguments</i>	257	21.2	<i>Textual substitution</i>	283
18.2	<i>Const references</i>	257	21.2.1	Dynamic definition of macros	283
18.2.1	Const references in range-based loops	259	21.2.2	Parametrized macros	284
18.3	<i>Const methods</i>	259	21.2.3	Type definitions	285
18.4	<i>Overloading on const</i>	260	21.3	<i>Conditionals</i>	285
18.5	<i>Const and pointers</i>	261	21.3.1	Disable a block of code	285
18.5.1	Old-style const pointers	262	21.3.2	Check for defined macros	285
18.6	<i>Mutable</i>	263	21.3.3	Numeric expression	286
18.7	<i>Compile-time constants</i>	265	21.3.4	Including a file only once	286
19	<b>Declarations and header files</b>	267	21.4	<i>Other preprocessor directives</i>	286
19.1	<i>Include files</i>	267	22	<b>Templates</b>	289
19.2	<i>Function declarations</i>	268	22.1	<i>Templated functions</i>	289
19.2.1	Separate compilation	269	22.2	<i>Templated classes</i>	290
19.2.2	Header files	269	22.2.1	Out-of-class method definitions	291
19.2.3	C and C++ headers	270	22.2.2	Specific implementations: template specializations	292
19.3	<i>Declarations for class methods</i>	271	22.2.3	Templates and separate compilation	292
19.4	<i>More</i>	271			
19.4.1	Header files and templates	271			
19.4.2	Namespaces and header files	271			

22.3	<i>Example: polynomials over fields</i>	293	24.8.3	Clocks	328
22.4	<i>Concepts</i>	295	24.8.4	Dates	330
23	<b>Error handling</b>	297	24.8.5	C mechanisms not to use anymore	330
23.1	<i>Assertions</i>	298	24.9	<i>File system</i>	330
23.2	<i>Exception handling</i>	299	24.10	<i>Enum classes</i>	330
23.2.1	Standard exceptions	300	24.11	<i>Orderings and the ‘spaceship’ operator</i>	331
23.2.2	Popular exceptions	300	25	<b>Concurrency</b>	335
23.2.3	Exception types	300	25.1	<i>Thread creation</i>	335
23.3	<i>Remaining topics</i>	302	25.1.1	Multiple threads	336
23.3.1	‘Where does this error come from’	302	25.1.2	<i>Asynchronous tasks</i>	337
23.3.2	Legacy mechanisms	303	25.1.3	Return results: futures and promises	338
24	<b>Standard Template Library</b>	305	25.1.4	The current thread	338
24.1	<i>Complex numbers</i>	305	25.1.5	More thread stuff	339
24.1.1	Complex support in C	306	25.2	<i>Data races</i>	339
24.2	<i>Limits</i>	306	25.3	<i>Synchronization</i>	340
24.3	<i>Containers</i>	307	26	<b>Obscure stuff</b>	343
24.3.1	Maps: associative arrays	308	26.1	<i>Auto</i>	343
24.3.2	Sets	309	26.1.1	<i>Declarations</i>	343
24.4	<i>Regular expression</i>	309	26.1.2	<i>decltype: declared type</i>	344
24.4.1	Regular expression syntax	311	26.2	<i>Casts</i>	345
24.5	<i>Tuples and structured binding</i>	311	26.2.1	Static cast	345
24.5.1	Example: roots of a quadratic equation	312	26.2.2	Dynamic cast	347
24.5.2	Example: square root	312	26.2.3	Const cast	348
24.6	<i>Union-like stuff: optionals, variants, expected</i>	314	26.2.4	Reinterpret cast	348
24.6.1	Tuples	314	26.2.5	A word about void pointers	349
24.6.2	Optional	315	26.3	<i>Fine points of scalar types</i>	349
24.6.3	Expected	317	26.3.1	Booleans	349
24.6.4	Variant	318	26.3.2	Integers	349
24.6.5	Any	321	26.3.3	Floating point types	351
24.7	<i>Random numbers</i>	322	26.3.4	Not-a-number	352
24.7.1	Generator	322	26.3.5	Common numbers	352
24.7.2	Distributions	322	26.4	<i>Ivalue vs rvalue</i>	352
24.7.3	Usage scenarios	323	26.4.1	Conversion	353
24.7.4	Permutations	326	26.4.2	References	354
24.7.5	C random function	326	26.4.3	Rvalue references	354
24.8	<i>Time</i>	327	26.5	<i>Move semantics</i>	355
24.8.1	Time durations	327	26.6	<i>Graphics</i>	355
24.8.2	Time points	328	26.7	<i>Standards timeline</i>	355
			26.7.1	C++98/C++03	355
			26.7.2	C++11	356

26.7.3	<b>C++14</b>	356	30.7	<i>Bit operations</i>	377
26.7.4	<b>C++17</b>	356	30.8	<i>Commandline arguments</i>	378
26.7.5	<b>C++20</b>	357	30.9	<i>Fortran type kinds</i>	378
26.7.6	<b>C++23</b>	357	30.9.1	<i>Kind selection</i>	378
27	<b>Graphics</b>	359	30.9.2	<i>Range</i>	380
27.1	<b>SFML</b>	359	30.10	<i>Quick comparison Fortran vs C++</i>	
27.2	<i>VT100 cursor control</i>	359	30.10.1	<i>Statements</i>	381
28	<b>C++ for C programmers</b>	361	30.10.2	<i>Input/Output, or I/O as we say</i>	382
28.1	<i>I/O</i>	361	30.10.3	<i>Expressions</i>	382
28.2	<i>Arrays</i>	361	30.11	<i>Review questions</i>	383
28.2.1	<i>Vectors from C arrays</i>	361	31	<b>Conditionals</b>	385
28.3	<i>Dynamic storage</i>	362	31.1	<i>Forms of the conditional statement</i>	385
28.4	<i>Strings</i>	362	31.2	<i>Operators</i>	385
28.5	<i>Pointers</i>	363	31.3	<i>Select statement</i>	386
28.5.1	<i>Parameter passing</i>	363	31.4	<i>Boolean variables</i>	386
28.5.2	<i>Addresses</i>	363	31.5	<i>Obsolete conditionals</i>	387
28.6	<i>Objects</i>	363	31.6	<i>Review questions</i>	387
28.7	<i>Namespaces</i>	363	32	<b>Loop constructs</b>	389
28.8	<i>Templates</i>	363	32.1	<i>Loop types</i>	389
28.9	<i>Obscure stuff</i>	363	32.2	<i>Interruptions of the control flow</i>	390
28.9.1	<i>Lambda</i>	363	32.3	<i>Implied do-loops</i>	391
28.9.2	<i>Const</i>	364	32.4	<i>Obsolete loop statements</i>	391
28.9.3	<i>Lvalue and rvalue</i>	364	32.5	<i>Review questions</i>	391
29	<b>C++ review questions</b>	365	33	<b>Procedures</b>	393
29.1	<i>Arithmetic</i>	365	33.1	<i>Subroutines and functions</i>	393
29.2	<i>Looping</i>	365	33.2	<i>Return results</i>	396
29.3	<i>Functions</i>	365	33.2.1	<i>The ‘result’ keyword</i>	398
29.4	<i>Vectors</i>	366	33.2.2	<i>The ‘contains’ clause</i>	398
29.5	<i>Vectors</i>	366	33.3	<i>Arguments</i>	399
29.6	<i>Objects</i>	366	33.3.1	<i>Keyword and optional arguments</i>	400
<b>III Fortran</b>		369	33.4	<i>Types of procedures</i>	401
30	<b>Basics of Fortran</b>	371	33.5	<i>Local variable save-ing</i>	401
30.1	<i>Source format</i>	371	34	<b>Scope</b>	403
30.2	<i>Compiling Fortran</i>	371	34.1	<i>Scope</i>	403
30.3	<i>Main program</i>	373	34.1.1	<i>Variables local to a program unit</i>	403
30.3.1	<i>Program structure</i>	373			
30.3.2	<i>Statements</i>	373			
30.3.3	<i>Comments</i>	374			
30.4	<i>Variables</i>	374			
30.4.1	<i>Declarations</i>	375			
30.4.2	<i>Initialization</i>	376			
30.5	<i>Complex numbers</i>	376			
30.6	<i>Expressions</i>	377			

34.1.2	Variables in an internal procedure	404	39.6.3	Restricting with where	436
35	<b>String handling</b>	405	39.6.4	Global condition tests	436
35.1	<i>String denotations</i>	405	39.7	<i>Array operations</i>	437
35.2	<i>Characters</i>	405	39.7.1	Loops without looping	437
35.3	<i>Strings</i>	405	39.7.2	Loops without dependencies	438
35.4	<i>Conversions</i>	406	39.7.3	Loops with dependencies	439
35.4.1	Character conversions	406	39.8	<i>Review questions</i>	440
35.4.2	String conversions	407	40	<b>Pointers</b>	441
35.5	<i>Further notes</i>	407	40.1	<i>Basic pointer operations</i>	441
36	<b>Structures, eh, types</b>	409	40.2	<i>Combining pointers</i>	442
36.1	<i>Derived type basics</i>	409	40.3	<i>Pointer status</i>	444
36.2	<i>Derived types and procedures</i>	410	40.4	<i>Pointers and arrays</i>	445
36.3	<i>Parameterized types</i>	411	40.5	<i>Example: linked lists</i>	445
37	<b>Modules</b>	413	40.5.1	Type definitions	446
37.1	<i>Modules for program modularization</i>	414	40.5.2	Attach a node at the end	447
37.2	<i>Module definition</i>	414	40.5.3	Insert a node in sort order	448
37.3	<i>Separate compilation</i>	415	41	<b>Input/output</b>	449
37.4	<i>Access</i>	416	41.1	<i>Types of I/O</i>	449
37.5	<i>Polymorphism</i>	416	41.2	<i>Print to terminal</i>	449
37.6	<i>Operator overloading</i>	417	41.2.1	Print on one line	449
38	<b>Classes and objects</b>	419	41.2.2	Printing arrays	450
38.1	<i>Classes</i>	419	41.3	<i>Formatted I/O</i>	450
38.1.1	Final procedures: destructors	421	41.3.1	Format letters	450
38.2	<i>Inheritance</i>	422	41.3.2	Repeating and grouping	451
38.3	<i>Operator overloading</i>	423	41.4	<i>File and stream I/O</i>	453
39	<b>Arrays</b>	425	41.4.1	Units	453
39.1	<i>Static arrays</i>	425	41.4.2	Other write options	453
39.1.1	<i>Initialization</i>	426	41.5	<i>Conversion to/from string</i>	454
39.1.2	<i>Array sections</i>	426	41.6	<i>Unformatted output</i>	454
39.1.3	<i>Integer arrays as indices</i>	428	41.7	<i>Print to printer</i>	455
39.2	<i>Multi-dimensional</i>	428	42	<b>Leftover topics</b>	457
39.2.1	<i>Querying an array</i>	430	42.1	<i>Interfaces</i>	457
39.2.2	<i>Reshaping</i>	431	42.1.1	Polymorphism	457
39.3	<i>Arrays to subroutines</i>	431	42.2	<i>Random numbers</i>	458
39.4	<i>Allocatable arrays</i>	432	42.3	<i>Timing</i>	458
39.4.1	<i>Returning an allocated array</i>	433	42.4	<i>Fortran standards</i>	459
39.5	<i>Array output</i>	434	43	<b>Fortran review questions</b>	461
39.6	<i>Operating on an array</i>	434			
39.6.1	<i>Arithmetic operations</i>	434			
39.6.2	<i>Intrinsic functions</i>	434			

---

43.1	<i>Fortran versus C++</i>	461	46.3	<i>Using one class in another</i>	481
43.2	<i>Basics</i>	461	46.4	<i>Is-a relationship</i>	483
43.3	<i>Arrays</i>	461	46.5	<i>Pointers</i>	483
43.4	<i>Subprograms</i>	462	46.6	<i>Operator overloading</i>	484
			46.7	<i>More stuff</i>	485
IV	<b>Exercises and projects</b>	463	47	<b>Zero finding</b>	487
44	<b>Style guide for project submissions</b>	465	47.1	<i>Root finding by bisection</i>	487
44.1	<i>General approach</i>	465	47.1.1	<i>Simple implementation</i>	487
44.2	<i>Style</i>	465	47.1.2	<i>Polynomials</i>	488
44.3	<i>Structure of your writeup</i>	465	47.1.3	<i>Left/right search points</i>	489
44.3.1	<i>Introduction</i>	466	47.1.4	<i>Root finding</i>	491
44.3.2	<i>Detailed presentation</i>	466	47.1.5	<i>Object implementation</i>	491
44.3.3	<i>Discussion and summary</i>	466	47.2	<i>Newton's method</i>	492
44.4	<i>Experiments</i>	466	47.2.1	<i>Function implementation</i>	492
44.5	<i>Detailed presentation of your work</i>	466	47.2.2	<i>Using lambda expressions</i>	493
44.5.1	<i>Presentation of numerical results</i>	466	47.2.3	<i>Templated implementation</i>	495
44.5.2	<i>Code</i>	467	48	<b>Eight queens</b>	497
45	<b>Prime numbers</b>	469	48.1	<i>Problem statement</i>	497
45.1	<i>Arithmetic</i>	469	48.2	<i>Solving the eight queens problem, basic approach</i>	498
45.2	<i>Conditionals</i>	469	48.3	<i>Developing a solution by TDD</i>	498
45.3	<i>Looping</i>	470	48.4	<i>The recursive solution method</i>	501
45.4	<i>Functions</i>	470	49	<b>Infectious disease simulation</b>	505
45.5	<i>While loops</i>	471	49.1	<i>Model design</i>	505
45.6	<i>Classes and objects</i>	471	49.1.1	<i>Other ways of modeling</i>	506
45.6.1	<i>Optimization</i>	473	49.2	<i>Coding</i>	506
45.6.2	<i>Exceptions</i>	473	49.2.1	<i>Person basics</i>	506
45.6.3	<i>Prime number decomposition</i>	473	49.2.2	<i>Interaction</i>	507
45.7	<i>Ranges</i>	474	49.2.3	<i>Population</i>	508
45.8	<i>Other</i>	474	49.3	<i>Epidemic simulation</i>	508
45.9	<i>Eratosthenes sieve</i>	475	49.3.1	<i>No contact</i>	509
45.9.1	<i>Arrays implementation</i>	475	49.3.2	<i>Contagion</i>	509
45.9.2	<i>Streams implementation</i>	475	49.3.3	<i>Vaccination</i>	510
45.10	<i>Range implementation</i>	476	49.3.4	<i>Spreading</i>	510
45.11	<i>User-friendliness</i>	477	49.3.5	<i>Mutation</i>	511
46	<b>Geometry</b>	479			
46.1	<i>Basic functions</i>	479			
46.2	<i>Point class</i>	479			

49.3.6	Diseases without vaccine: Ebola and Covid-19	512	53.1	<i>Mathematical preliminaries</i>	537
49.4	<i>Ethics</i>	512	53.2	<i>Matrix storage</i>	538
49.5	<i>Bonus: parallelism</i>	512	53.2.1	<i>Submatrices</i>	540
49.6	<i>Bonus: testing</i>	513	53.3	<i>Multiplication</i>	541
49.7	<i>Bonus: mathematical analysis</i>	513	53.3.1	<i>One level of blocking</i>	541
49.8	<i>Project writeup and submission</i>	513	53.3.2	<i>Recursive blocking</i>	541
49.8.1	Program files	513	53.4	<i>Performance issues</i>	541
49.8.2	Writeup	514	53.5	<i>Bonus: parallelism</i>	542
50	<b>Google PageRank</b>	515	53.6	<i>Bonus: optimal performance</i>	542
50.1	<i>Basic ideas</i>	515	54	<b>The Great Garbage Patch</b>	543
50.2	<i>Clicking around</i>	516	54.1	<i>Problem and model solution</i>	543
50.3	<i>Graph algorithms</i>	517	54.2	<i>Program design</i>	543
50.4	<i>Page ranking</i>	517	54.2.1	<i>Grid update</i>	544
50.5	<i>Graphs and linear algebra</i>	518	54.3	<i>Testing</i>	544
50.6	<i>Bonus: parallelism</i>	519	54.3.1	<i>Animated graphics</i>	544
51	<b>Redistricting</b>	521	54.4	<i>Modern programming techniques</i>	546
51.1	<i>Basic concepts</i>	521	54.4.1	<i>Object oriented programming</i>	546
51.2	<i>Basic functions</i>	522	54.4.2	<i>Data structure</i>	546
51.2.1	Voters	522	54.4.3	<i>Cell types</i>	546
51.2.2	Populations	522	54.4.4	<i>Ranging over the ocean</i>	546
51.2.3	Districting	523	54.4.5	<i>Random numbers</i>	547
51.3	<i>Strategy</i>	524	54.5	<i>Explorations</i>	547
51.4	<i>Efficiency: dynamic programming</i>	526	54.5.1	<i>Code efficiency</i>	547
51.5	<i>Extensions</i>	526	55	<b>Graph algorithms</b>	549
51.6	<i>Ethics</i>	527	55.1	<i>Traditional algorithms</i>	549
52	<b>Amazon delivery truck scheduling</b>	529	55.1.1	<i>Code preliminaries</i>	549
52.1	<i>Problem statement</i>	529	55.1.2	<i>Level set algorithm</i>	551
52.2	<i>Coding up the basics</i>	529	55.1.3	<i>Dijkstra's algorithm</i>	551
52.2.1	Address list	529	55.2	<i>Linear algebra formulation</i>	552
52.2.2	Add a depot	532	55.2.1	<i>Code preliminaries</i>	552
52.2.3	Greedy construction of a route	532	55.2.2	<i>Unweighted graphs</i>	553
52.3	<i>Optimizing the route</i>	533	55.2.3	<i>Dijkstra's algorithm</i>	554
52.4	<i>Multiple trucks</i>	534	55.2.4	<i>Sparse matrices</i>	554
52.5	<i>Amazon prime</i>	536	55.3	<i>Bonus: parallelism</i>	554
52.6	<i>Dynamicism</i>	536	55.4	<i>Further explorations</i>	554
52.7	<i>Bonus: parallelism</i>	536	55.5	<i>Tests and reporting</i>	555
52.8	<i>Ethics</i>	536	56	<i>Congestion</i>	557
53	<b>High performance linear algebra</b>	537	56.1	<i>Problem statement</i>	557

56.2	<b>Code design</b>	557	63.1.1	Using an external library	583
56.2.1	Cars	557	63.1.2	Obtaining and installing an external library	584
56.2.2	Street	558	63.2	<i>Options processing: cxxopts</i>	585
56.2.3	Unit tests	558	63.2.1	Traditional commandline parsing	586
57	<b>DNA Sequencing</b>	559	63.2.2	The <code>cxxopts</code> library	586
57.1	<i>Basic functions</i>	559	63.2.3	Install and usage	588
57.2	<i>De novo shotgun assembly</i>	559	63.3	<i>Linear algebra libraries</i>	589
57.2.1	Overlap layout consensus	560	64	<b>Programming strategies</b>	591
57.2.2	De Bruijn graph assembly	560	64.1	<i>A philosophy of programming</i>	591
57.3	<i>'Read' matching</i>	560	64.2	<i>Programming: top-down versus bottom up</i>	591
57.3.1	Naive matching	560	64.2.1	Worked out example	592
57.3.2	Boyer-Moore matching	560	64.3	<i>Coding style</i>	593
57.4	<i>Bloom filters</i>	562	64.4	<i>Documentation</i>	593
58	<b>Memory allocation</b>	563	64.5	<i>Best practices: C++ Core Guidelines</i>	593
59	<b>Ballistics calculations</b>	565	65	<b>Performance optimization</b>	595
59.1	<i>Introduction</i>	565	65.1	<i>Problem statement</i>	595
59.1.1	Physics	567	65.2	<i>Coding</i>	595
59.1.2	Numerical analysis	568	65.2.1	Optimization: save on allocation	597
60	<b>Cryptography</b>	569	65.2.2	Caching in a static vector	598
60.1	<i>The basics</i>	569	65.3	<i>Vector vs array</i>	598
60.2	<i>Cryptography</i>	569	66	<b>Tiniest of introductions to algorithms and data structures</b>	601
60.3	<i>Blockchain</i>	569	66.1	<i>Data structures</i>	601
61	<b>Climate change</b>	571	66.1.1	Stack	601
61.1	<i>Reading the data</i>	571	66.1.2	Linked lists	601
61.2	<i>Statistical hypothesis</i>	571	66.1.3	Trees	609
62	<b>Desk Calculator</b>		66.1.4	Other graphs	611
	<b>Interpreter</b>	573	66.2	<i>Algorithms</i>	612
62.1	<i>Named variables</i>	573	66.2.1	Sorting	612
62.2	<i>First modularization</i>	574	66.2.2	Graph algorithms	613
62.3	<i>Event loop and stack</i>	574	66.3	<i>Programming techniques</i>	614
62.3.1	Stack	575	66.3.1	Memoization	614
62.3.2	Stack operations	575			
62.3.3	Item duplication	577			
62.4	<i>Modularizing</i>	578			
62.5	<i>Object orientation</i>	579			
62.5.1	Operator overloading	579			
V	<b>Advanced topics</b>	581			
63	<b>External libraries</b>	583			
63.1	<i>What are software libraries?</i>	583			

67	<b>Provably correct programs</b>	617	69.2	<i>Example: integer overflow</i>	638
67.1	<i>Loops as quantors</i>	617	69.3	<i>More gdb</i>	639
67.1.1	Forall-quantor	617	69.3.1	Run with commandline arguments	639
67.1.2	Thereis-quantor	618	69.3.2	Source listing and proper compilation	639
67.1.3	Quantors through ranges	619	69.3.3	Stepping through the source	640
67.2	<i>Predicate proving</i>	620	69.3.4	Inspecting values	641
67.3	<i>Flame</i>	621	69.3.5	A NaN example	642
67.3.1	Derivation of the common algorithm	622	69.3.6	Assertions	644
68	<b>Unit testing and Test-Driven Development</b>	627	70	<b>Complexity</b>	645
68.1	<i>Types of tests</i>	627	70.1	<i>Theory</i>	645
68.2	<i>Unit testing frameworks</i>	628	70.2	<i>Time complexity</i>	645
68.2.1	Test cases	628	70.3	<i>Space complexity</i>	645
68.3	<i>Example: zero-finding by bisection</i>	631	71	<b>Support tools</b>	647
68.4	<i>An example: quadratic equation roots</i>	631	71.1	<i>Editors and development environments</i>	647
68.5	<i>Eight queens example</i>	633	71.2	<i>Compilers</i>	647
68.6	<i>Practical aspects of using Catch2</i>	633	71.3	<i>Build systems</i>	647
68.6.1	Installing Catch2	633	71.4	<i>Debuggers</i>	647
68.6.2	Two usage modes	634	VI	Index and such	649
68.6.3	Compilation	634		General index of terms	651
69	<b>Debugging with gdb</b>	637	72	<b>Index of C++ keywords</b>	659
69.1	<i>A simple example</i>	637	73	<b>Index of Fortran keywords</b>	665
69.1.1	Invoking the debugger	637	74	<b>Bibliography</b>	669

## **PART I**

### **INTRODUCTION**



# Chapter 1

## Introduction

### 1.1 Programming and computational thinking

In this chapter we take a look at the history of computers and computer programming, and think a little about what programming involves.

#### 1.1.1 History

In the early days of computing, hardware design was seen as challenging, while programming was little more than data entry. The fact that one of the earliest programming languages was called ‘Fortran’, for



Figure 1.1: Robert Oppenheimer and John von Neumann

‘formula translation’, speaks to this: once you have the math, programming was thought to be nothing more than translating the math into code. The fact that programs could have subtle errors, or *bugs*, came as quite a surprise to the earliest computer designers.

The fact that programming was not as highly valued also had the side-effect that many of the early programmers were women. Before electronic computers, a ‘computer’ was a person executing computations, probably with a mechanical calculating device, and often these were women. From this, the earliest people programming electronic computers to perform these calculations were, usually mathematically educated, women. Two famous examples were Navy Rear-admiral Grace Hopper, inventor of the Cobol language, and Margaret Hamilton who led the development of the Apollo program software. This situation changed after the 1960s and certainly with the advent of PCs<sup>1</sup>.

---

1. <http://www.sysgen.com.ph/articles/why-women-stopped-coding/27216>

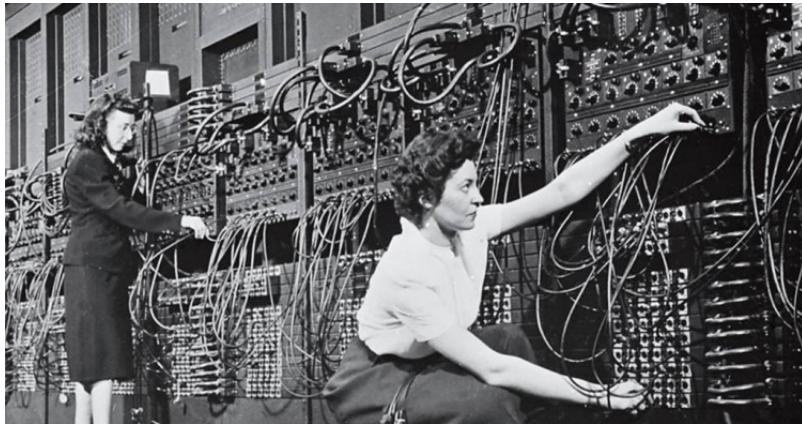


Figure 1.2: Programming the ENIAC

### 1.1.2 Is programming science, art, or craft?

As the previous section argued, programming is more than simple translation of math into instructions for hardware. Could it be a science? There are certainly scientific aspects to programming:

- Algorithms and complexity theory have a lot of math in them.
- Programming language design is another mathematically tinged subject.

However, programming itself is not a science.

The term ‘software engineering’ may lead you to suspect that designing and producing software is an engineering discipline, but this is also not quite the case. There is no certification for software engineers, and there is no body of accepted techniques the way there is for civil engineering and such disciplines.

For a large part programming is a discipline. What constitutes a good program is a matter of taste. That does not mean that there aren’t recommended practices. In this course we will emphasize certain practices that we think lead to good code, as likewise will discourage you from certain idioms.

None of this is an exact science. There are multiple programs that give the right output. However, programs are rarely static. They often need to be amended or extended, or even fixed, if erroneous behavior comes to light, and in that case a badly written program can be a detriment to programmer productivity. An important consideration, therefore, is intelligibility of the program, to another programmer, to your professor in this course, or even to yourself two weeks from now.

### 1.1.3 Computational thinking

Mathematical thinking:

- Number of people per day, speed of elevator  $\Rightarrow$  yes, it is possible to get everyone to the right floor.
- Distribution of people arriving etc.  $\Rightarrow$  average wait time.

Sufficient condition  $\neq$  existence proof.

Computational thinking: actual design of a solution

- Elevator scheduling: someone at ground level presses the button, there are cars on floors 5 and 10; which one do you send down?

Coming up with a strategy takes creativity!

**Exercise 1.1.** A straightforward calculation is the simplest example of an algorithm.

Calculate how many schools for hair dressers the US can sustain. Identify the relevant factors, estimate their sizes, and perform the calculation.

**Exercise 1.2.** Algorithms are usually not uniquely determined:  
there is more than one way solve a problem.

Four self-driving cars arrive simultaneously at an all-way-stop intersection. Come up with an algorithm that a car can follow to safely cross the intersection. If you can come up with more than one algorithm, what happens when two cars using different algorithms meet each other?

Looking up a name in the phone book

- start on page 1, then try page 2, et cetera
- or start in the middle, continue with one of the halves.

What is the average search time in the two cases?

Having a correct solution is not enough!

A powerful programming language serves as a framework within which we organize our ideas. Every programming language has three mechanisms for accomplishing this:

- primitive expressions
- means of combination
- means of abstraction

*Abelson and Sussman, The Structure and Interpretation of Computer Programs*

- The elevator programmer probably thinks: ‘if the button is pressed’, not ‘if the voltage on that wire is 5 Volt’.
- The Google car programmer probably writes: ‘if the car before me slows down’, not ‘if I see the image of the car growing’.
- ... but probably another programmer had to write that translation.

A program has layers of abstractions.

Abstraction means your program talks about your application concepts, rather than about numbers and characters and such.

Your program should read like a story about your application; not about bits and bytes.

## 1. Introduction

---

Good programming style makes code intelligible and maintainable.

(Bad programming style may lead to lower grade.)

The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague — Edsger Dijkstra

What is the structure of the data in your program?

Stack: you can only get at the top item



Queue: items get added in the back, processed at the front



A program contains structures that support the algorithm. You may have to design them yourself.

### 1.1.4 Hardware

Yes, it's there, but we don't think too much about it in this course.

Advanced programmers know that hardware influences the speed of execution; see HPC book [13], section ??.

### 1.1.5 Algorithms

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time

[A. Levitin, Introduction to The Design and Analysis of Algorithms,  
Addison-Wesley, 2003]

The instructions are written in some language:

- We will teach you C++ and Fortran;
- the compiler translates those languages to machine language

- Simple instructions: arithmetic.
- Complicated instructions: control structures
  - conditionals

– loops

- Input and output data: to/from file, user input, screen output, graphics.
- Data during the program run:
  - Simple variables: character, integer, floating point
  - Arrays: indexed set of characters and such
  - Data structures: trees, queues
    - \* Defined by the user, specific for the application
    - \* Found in a library (big difference between C/C++!)

## 1.2 About the choice of language

There are many programming languages, and not every language is equally suited for every purpose. In this book you will learn C++ and Fortran, because they are particularly good for scientific computing. And by ‘good’, we mean

- They can express the sorts of problems you want to tackle in scientific computing, and
- they execute your program efficiently.

There are other languages that may not be as convenient or efficient in expressing scientific problems. For instance, *python* is a popular language, but typically not the first choice if you’re writing a scientific program. As an illustration, here is simple sorting algorithm, coded in both C++ and python.

Python vs C++ on bubblesort:

```
for i in range(n-1):
    for j in range(n-i-1):
        if numbers[j+1]<numbers[j]:
            swaptmp = numbers[j+1]
            numbers[j+1] = numbers[j]
            numbers[j] = swaptmp

for (int i=0; i<n-1; i++)
    for (int j=0; j<n-1-i; j++)
        if (numbers[j+1]<numbers[j]) {
            int swaptmp = numbers[j+1];
            numbers[j+1] = numbers[j];
            numbers[j] = swaptmp;
        }

$ python bubblesort.py 5000
Elapsed time: 12.1030311584
$ ./bubblesort 5000
Elapsed time: 0.24121
```

But this ignores one thing: the sorting algorithm we just implemented is not actually a terribly good one, and in fact python has a better one built-in.

Python with quicksort algorithm:

```
numpy.sort(numbers, kind='quicksort')

[] python arraysort.py 5000
Elapsed time: 0.00210881233215
```

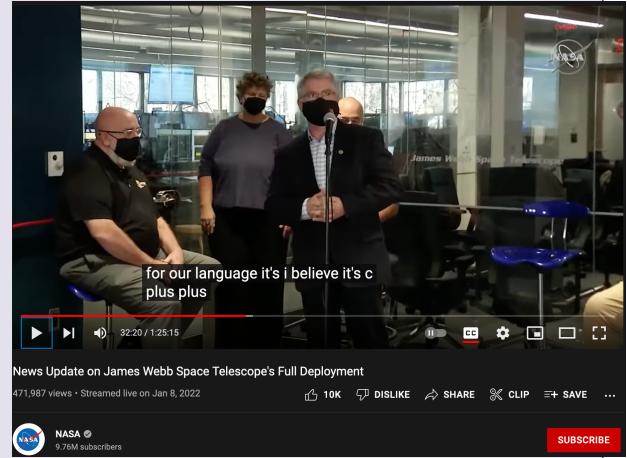
## 1. Introduction

---

So that is another consideration when choosing a language: is there a language that already comes with the tools you need. This means that your application may dictate the choice of language. If you're stuck with one language, don't reinvent the wheel! If someone has already coded it or it's part of the language, don't redo it yourself.

Application domains where C++ rules:

- Scientific computing;  
interoperability with C/Python code.
- Embedded processors
- Game engines



# Chapter 2

## Logistics

### 2.1 Programming environment

Programming can be done in any number of ways. It is possible to use an Integrated Development Environment (IDE) such as *Xcode* or *Visual Studio*, but for if you're going to be doing some computational science you should really learn a *Unix* variant.

- If you have a *Linux* computer, you are all set.
- If you have an *Apple* computer, it is easy to get you going. You can use the standard *Terminal* program, or you can use a full *X windows* installation, such as *XQuartz*, which makes Unix graphics possible. This, and other Unix programs can be obtained through a *package manager* such as *homebrew* or *macports*.
- *Microsoft Windows* users can use *putty* but it is probably a better solution to install a virtual environment such as *VMware* (<http://www.vmware.com/>) or *Virtualbox* (<https://www.virtualbox.org/>).

Next, you should know a text editor. The two most common ones are *vi* and *emacs*.

#### 2.1.1 Language support in your editor

The two most popular editors are *emacs* and *vi* or *vim*. Both have support for programming languages, doing syntax coloring, and helping you with the correct indentation.. Most of the time, your editor will detect what language a file is written in, based on the file extension:

- `cxx, cpp, cc` for C++, and
- `f90, F90` for Fortran.

If your editor somehow doesn't detect the language, you can add a line at the top of the file:

```
// -*- c++ -*-
```

for C++ mode, and

```
! -*- f90 -*-
```

for Fortran mode.

Main advantages are automatic indentation (C++ and Fortran) and supplying block end statements (Fortran). The editor will also apply 'syntax coloring' to indicate the difference between keywords and variables.

## 2.2 Compiling

The word ‘program’ is ambiguous. Part of the time it means the *source code*: the text that you type, using a text editor. And part of the time it means the *executable*, a totally unreadable version of your source code, that can be understood and executed by the computer. The process of turning your source code into an executable is called *compiling*, and it requires something called a *compiler*. (So who wrote the source code for the compiler? Good question.)

Here is the workflow for program development

1. You think about how to solve your program
2. You write code using an editor. That gives you a source file.
3. You compile your code. That gives you an executable.  
Oh, make that: you try to compile, because there will probably be compiler errors: places where you sin against the language syntax.
4. You run your code. Chances are it will not do exactly what you intended, so you go back to the editing step.

## 2.3 Your environment

The following exercise is for the situation where there is a central class computer. To prove that you have your connectivity sorted out, do the following.

**Exercise 2.1.** Do an online search into the history of computer programming. Write a page, if possible with illustration, and turn this into a pdf file. Submit this to your teacher.

# Chapter 3

## Teachers guide

This book was written for a one-semester introductory programming course at The University of Texas at Austin, aimed primarily at students in the physical and engineering sciences. Thus, examples and exercises are as much as possible scientifically motivated. This target audience also explains the inclusion of Fortran.

This book is not encyclopedic. Rather than teaching each topic in its full glory, the author has taken a ‘recommended practices’ approach, where students learn enough of each topic to become a competent programmer. (Recommended by who you might ask? The author freely admits being guided by his own taste. However, he lets himself be informed by plenty of other current literature.) This serves to keep this book at a manageable length, and to minimize class lecture time, emphasizing lab exercises instead.

Even then, there is more material here than can be covered and practiced in one semester. If only C++ is taught, it is probably possible to cover the whole of Part II; for the case where both C++ and Fortran are taught, we have a suggested time line below.

### 3.1 Justification

The chapters of Part II and Part III are presented in suggested teaching order. Here we briefly justify our (non-standard) sequencing of topics and outline a timetable for material to be covered in one semester. Most notably, Object-Oriented programming is covered before arrays, and pointers come very late, if at all.

There are several thoughts behind this. For one, dynamic arrays in C++ are most easily realized through the `std::vector` mechanism, which requires an understanding of classes. The same goes for `std::string`.

Secondly, in the traditional approach, object-oriented techniques are taught late in the course, after all the basic mechanisms, including arrays. We consider OOP to be an important notion in program design, and central to C++, rather than an embellishment on the traditional C mechanisms, so we introduce it as early as possible.

Even more elementary, we emphasize range-based loops as much as possible over indexed loops, since ranges are increasing in importance in recent language versions.

### 3.1.1 Algorithms

Some of the programming exercises in this course ask the student to reproduce algorithms that exist in the `std::algorithm` header. Thus this course could be open to the criticism that students should learn their Standard Template Library (STL) algorithms, rather than recreating them themselves. (See section 14.3.)

My defense would be that a programmer should know more than what algorithms to pick from the standard library. Students should understand the mechanisms behind these algorithms, and be able to reproduce them, so that, when this is needed, they can code variations of these algorithms.

## 3.2 Time line for a C++/F03 course

As remarked above, this book is based on a course that teaches both C++ and Fortran2003. Here we give the time line used, including some of the assigned exercises.

For a one semester course of slightly over three months, two months would be spent on C++ (see table 3.1), after which a month is enough to explain Fortran; see table 3.3. Remaining time will go to exams and elective topics.

### 3.2.1 Advanced topics

We also outline a ‘C++ 101.5’ course: somewhere between beginning and truly advanced. Here we assume that the student has had about 8 lectures worth of C++, covering

1. basic control structures,
2. simple functions including parameter passing by reference,
3. arrays through `std::vector`.

Based on this, the topics in table 3.2 can be taught in that order.

### 3.2.2 Project-based teaching

To an extent it is inevitable that students will do a number of exercises that are not connected to any earlier or later ones. However, to give some continuity, this book contains some programming projects that students gradually build towards.

The following are simple projects that have a sequence of exercises building on each other:

**Prime** Prime number testing, culminating in prime number sequence objects, and testing a corollary of the Goldbach conjecture. Chapter 45.

**Geom** Geometry related concepts; this is mostly an exercise in object-oriented programming. Chapter 46.

**Root** Numerical zero-finding methods. Chapter 47.

The following project are halfway serious research projects:

**Infect** The spreading of infectious diseases; these are exercises in object-oriented design. Students can explore various real-life scenarios. Chapter 49.

**Pagerank** The Google Pagerank algorithm. Students program a simulated internet, and explore pageranking, including ‘search engine optimization’. This exercise uses lots of pointers. Chapter 50.

**Gerrymandering** Redistricting through dynamic programming. Chapter 51.

lesson	Topic	Exercises				
		in-class	homework	prime	geom	infect
1	Statements and expres- sions	4.5	4.9 (T)	45.1		
2	Conditionals	5.2 (S), 5.3	5.4 (T)	45.2		
3	Looping	6.6 (S), 6.5	6.7 (T)	45.3, 45.4		
4	continue					
5	Functions	7.1 (S), 7.2	7.6	45.6, 45.7 (T)		
6	continue	7.11			46.1	
7	I/O		12.1			
8	Objects			9.8 (S), 45.8 (T), 45.10	46.3	49.1
9	continue					
10	has-a rela- tion				46.10 (T), 46.11, 46.1, 46.14	49.4
11	inheritance				46.17, 46.18	
12	Vectors	10.1 (S), 10.3	10.22	45.19		49.4 and further
13	continue					
14	Strings					

Table 3.1: Two-month lesson plan for C++; the annotation ‘(S)’ indicates that a skeleton code is available; ‘(T)’ indicates that a tester script is available.

### 3. Teachers guide

---

lesson	Topic	Book	Prerequisite	Exercises	
				In-class	Homework
1	Welcome, accounts				Essay, coe_history
2,3	Unix, compilation, check prior knowledge				Collatz: 6.14; swap: 7.5; vectors: 10.20, coe_catchup
4, 5	Test-driven development	68	Separate compilation 19	68.1 (S), 47.1–47.6	47.8 coe_bisection
6, 7	Objects	9		9.3 (S), 9.8 (S)	45.8 coe_primes
8	Class inclusion	9.2		46.10, 46.14	
9	Inheritance	9.3		46.17	45.9, 45.10 coe_goldbach
10	Vectors	10		10.1 (S), 10.9, 10.10	
11	Vectors in classes	10.6		10.15 (S)	10.22 coe_pascal
12, 13	Lambda functions	13		47.10 (S), 47.11	47.12, 47.13 coe_newton
14,15	STL, variant, optional	24.6.2		45.18	Eight queens: 48
16	Pointers	16, 66.1.2			66.3–66.6
17	C pointers	17			
18	libraries and cmake cxxopts fmt random Exceptions formal approaches		63.1	63.2	

Table 3.2: Advanced lessons for C++; the annotation ‘(S)’ indicates that a skeleton code is available; ‘(T)’ indicates that a tester script is available.

lesson	Topic	Book	Slides	in-class	homework
1	Statements and expres- sions	30	4F		
	Conditionals	31	5F		
2	Looping	32	6F		32.1
	Functions	33	7F		
3	I/O	41	8F		
	Arrays	39	14F	39.1, 39.3, 39.5	
4	Objects	38	10F	38.2	
	Modules	37	9F (?)		37.2
5	Pointers	40	15F		

Table 3.3: Accelerated lesson plan for Fortran; the annotation ‘(S)’ indicates that a skeleton code is available; ‘(T)’ indicates that a tester script is available.

**Scheduling** Exploration of concepts related to the Multiple Traveling Salesman Problem (MTSP), modeling *Amazon Prime*. Chapter 52.

**Lapack** Exploration of high performance linear algebra. Chapter 53.

Rather than including the project exercises in the didactic sections, each section of these projects list the prerequisite basic sections.

The project assignments give a fairly detailed step-by-step suggested approach. This acknowledges the theory of Cognitive Load [15].

Our projects are very much computation-based. A more GUI-like approach to project-based teaching is described in [7].

### 3.2.3 Choice: Fortran or advanced topics

After two months of grounding in OOP programming in C++, the Fortran lectures and exercises reprise this sequence, letting the students do the same exercises in Fortran that they did in C++. However, array mechanisms in Fortran warrant a separate lecture.

## 3.3 Scaffolding

As much as possible, when introducing a new topic we try to present a working code, with the first exercise being a modification of this code. At the root level of the repository is a directory `skeletons`, containing the example programs. In the above tables, exercises for which a skeleton code is given are marked with ‘(S)’.

## 3.4 Grading guide

There is more to a program than getting the right output. It is sometimes said that a program is read more often than executed. Thus it needs to be written in a manner that it convinces the reader of its correctness.

### 3.4.1 A discipline of programming

In this section we outline a number of style points: a code using them may very well give the right answer, but they go against what is commonly considered good or clean code.

Here are some general guidelines on things that should count as negatives when grading.

The general guiding principle should be:

Code is primarily for humans to read, only secondarily for computers to execute.  
This means that in addition to being correct, the code has to convince the reader  
that the result is correct.

As a corollary:

Code should only be optimized when it is correct. Clever tricks detract from readability, and should only be applied when demonstrably needed.

In the chapters of this book we will also refer to the *Core Guidelines* for C++ <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines> for topic-specific guidelines.

### 3.4.2 Code layout and naming

Code should use proper indentation. Incorrect indentation deceives the reader.

Non-obvious code segments should be commented, but proper naming of variables and functions goes a long way towards making this less urgent.

### 3.4.3 Basic elements

- Variables should have descriptive names. For instance, `count` is not descriptive: half of all integers are used for counting. Count of what?
- Do not use global variables:

```
int i;
int main() {
    cin >> i;
}
```

### 3.4.4 Looping

The loop variable should be declared locally, in the loop header, if there is no overwhelming reason for declaring it global.

Local declaration:

```
for ( int i=0; i<N; i++) {
    // something with i
}
```

global:

```
int i;
for ( i=0; i<N; i++) {
    // something with i
    if (something) break;
}
// look at i to see where the break was
```

Use range-based loops if the index is not strictly needed.

### 3.4.5 Functions

There is no preference whether to define a function entirely before the main, or after the main with only its declaration before.

Defined before:

```
bool f( double x ) {
    return x>0;
}
int main() {
    cout << f(5.1);
}
```

Only declaration given before:

```
bool f(double x);
int main() {
    cout << f(5.1);
}
bool f( double x ) {
    return x>0;
}
```

Only use C++-style parameter passing by value or reference: do not use C-style `int*` parameters and such.

### 3.4.6 Scope

Variables should be declared in the most local scope that uses them. Declaring all variables at the start of a subprogram is needed in Fortran, but is to be discouraged in C++

### 3.4.7 Classes

- Do not declare data members `public`.
- Only write accessor functions if they are really needed.
- Make sure method names are descriptive of what the method does.
- The keyword `this` is hardly ever needed. This usually means the student has been looking at stackoverflow too much.

### 3.4.8 Arrays vs vectors

- Do not use old-style C arrays.  
`int indexes[5];`
- Certainly never use `malloc` or `new`.
- Iterator (`begin`, `erase`) are seldom used in this course, and should only be used if strictly needed, for instance with ‘algorithms’.

### 3.4.9 Strings

Do not use old-style C strings:

```
char *words = "and another thing";
```

### 3.4.10 Other remarks

#### 3.4.10.1 Uninitialized variables

Uninitialized variables can lead to undefined or indeterminate behavior. Bugs, in other words.

```

int i_solution;
int j_solution;
bool found_solution;
for(int i = 0; i < 10; i++) {
    for(int j = 0; j < 10; j++) {
        if(i*j > n) {
            i_solution = i;
            j_solution = j;
            found_solution = true;
            break;
        }
    }
    if(found_solution) {break; }
}
cout << i_solution << "," << j_solution << endl;

%% icpc -o missinginit missinginit.cpp && echo 40 | ./missinginit
0, -917009232
6, 7

```

This whole issue can be side stepped if the compiler or runtime system can detect this issue. Code structure will often prevent detection, but runtime detection is always possible, in principle.

For example, the Intel compiler can install a run-time check:

```

%% icpc -check=uninit -o missinginit missinginit.cpp && echo 40 | ./missi
Run-Time Check Failure: The variable 'found_solution' is being used in mi
Aborted (core dumped)

```

#### 3.4.10.2 Clearing objects and vectors

The following idiom is found regularly:

```

vector<int> testvector;
for ( /* possibilities */ ) {
    testvector.push_back( something );
    if ( testvector.sometest() )
        // remember this as the best
    testvector.clear();
}

```

A similar idiom occurs with classes, which students endow with a `reset()` method.

This is more elegantly done by declaring the `testvector` inside the loop: the reset is then automatically taken care of.

The general principle here is that entities need to be declared as local as possible.

## **PART II**

**C++**



## Chapter 4

### Basic elements of C++

#### 4.1 From the ground up: Compiling C++

In this chapter and the next you are going to learn the C++ language. But first we need some externalia: where do you get a program and how do you handle it?

In programming you have two kinds of files:

- *source files*, which are understandable to you, and which you create with an editor such as `vi` or `emacs`; and
- *binary files*, which are understandable to the computer, and unreadable to you.

Your source files are translated to binary by a *compiler*, which ‘compiles’ your source file.

Let’s look at an example:

```
icpc -o myprogram myprogram.cpp
```

This means:

- you have a source code file `myprogram.cpp`;
- and you want an executable file as result, called `myprogram`,
- and your compiler is the Intel compiler `icpc`. (If you want to use the C++ compiler of the `GNU` project you specify `g++`; the compiler of the `clang` project is `clang++`.)

Let’s do an example.

**Exercise 4.1.** Make a file `null.cpp` with the following lines:

```
// basic/null.cpp
#include <iostream>
using std::cout;

int main() {
    return 0;
}
```

and compile it. Intel compiler:

```
icpc -o nullprogram null.cpp
```

Run this program (it gives no output):

## 4. Basic elements of C++

---

```
./nullprogram
```

In the above program:

1. The first two lines are magic, for now. Always include them. Ok, if you want to know: the `#include` line is a *preprocessor* (chapter 21) directive; it includes a *header* into your program that makes certain functionality available.
2. The `main` line indicates where the program starts; between its opening and closing brace will be the *program statements*.
3. The `return` statement indicates successful completion of your program. (If you wonder about the details of this, see section 4.6.1.)

If you followed the above instructions, and as you may have guessed, you saw that this program produces absolutely no output when you run it.

Are you getting tired of typing the same command over and over again?

1. Up-arrow in the shell goes back through the *command history*;
2. You can type the first couple of characters of a name and use *tab completion*;
3. You can make a little shell script, let's call it `mycompile.sh`:

```
icpc -o myprogram -O2 -g myprogram.cpp
```

and then

```
source mycompile.sh
```

4. You can learn about *Make* and *CMake*.

If you do `ls` in your current directory, you see that you now have two files: the source file `zero.cc` and the executable `zeroprogram`. You have some freedom in choosing these names.

File names can have extensions: the part after the dot. (The part before the dot is completely up to you.)

- `program.cpp` or `program.cc` or `program.cxx` are typical extensions for C++ sources.
- `program.cpp` has a possible possible confusion with ‘C PreProcessor’, but it seems to be the standard, so we will use it in this course.
- Using `program` without extension usually indicates an *executable*. (What happens if you leave the `-o myprogram` part off the compile line?)

Let's make the program do something: display a ‘hello world’ message on your screen. For now, just copy this line; the details of what it all means will come later.

**Exercise 4.2.** Add this line:

```
// basic/hello.cpp  
cout << "Hello world!" << '\n';
```

(copying from the pdf file is dangerous! please type it yourself)

Compile and run again. What is the output?

Test your knowledge of the file types involved in programming!

**Review 4.1.** True or false?

1. The programmer only writes source files, no binaries.
2. The computer only executes binary files, no human-readable files.

### 4.1.1 A quick word about unix commands

The compile line

```
g++ -o myprogram myprogram.cpp
```

can be thought of as consisting of three parts:

- The command *g++* that starts the line and determines what is going to happen;
- The argument *myprogram.cpp* that ends the line is the main thing that the command works on; and
- The option/value pair *-o myprogram*. Most Unix commands have various options that are, as the name indicates, optional. For instance you can tell the compiler to try very hard to make a fast program:

```
g++ -O3 -o myprogram myprogram.cpp
```

Options can appear in any order, so this last command is equivalent to

```
g++ -o myprogram -O3 myprogram.cpp
```

Be careful not to mix up argument and option. If you type

```
g++ -o myprogram.cpp myprogram
```

then Unix will reason: ‘*myprogram.cpp* is the output, so if that file already exists (which, yes, it does) let’s just empty it before we do anything else’. And you have just lost your program. Good thing that editors like *emacs* keep a backup copy of your file.

### 4.1.2 Build environments

Above we didn’t say anything about how to make your sources files. Also, while you saw a compile line, you may not need to issue those by hand. It all depends on how sophisticated your build environment is. Here we give some possibilities, but this course leaves the choice up to you.

#### 4.1.2.1 Commandline and editor

The traditional way of developing software is by using an editor – such as *emacs*, *vim*, *nano* – and typing compile commands on the commandline of Unix or some other Operating System (OS).

If you are proficient in Unix, this is not a bad strategy. You can make life easier by using *Make* or *CMake* but that’s mostly for more complicated programs than you will run into in the first number of chapters of this course.

#### 4.1.2.2 Integrated build system

There are some really nice programs with a graphical user interface that make software development less painful if you're not such a Unix-head. They help you with editing your file, and compiling is a press-on-a-button.

Specific to Apple, there is *XCode*, and commercially from Microsoft there is *Visual Studio*. The latter has a free (and limited) version *VSCode*. Another commercial product widely in use is *CLion*.

Finally, there is the open source *Eclipse* software.

#### 4.1.3 C++ is a moving target

The C++ language has gone through a number of standards. (This is described in some detail in section 26.7.) In this course we focus on recent standards: C++20, and to some amount C++23. Your compiler may assume an earlier standard by default, so constructs taught here may be marked as ungrammatical.

You can tell your compiler to use a modern standard:

```
icpx -std=c++17 [other options]
```

but to save yourself a lot of typing, you can define

```
alias icpx='icpx -std=c++17'
```

in your shell startup files. On the class *isp* machine this alias has been defined by default.

## 4.2 Statements

Each programming language has its own (very precise!) rules for what can go in a source file. Globally we can say that a program contains instructions for the computer to execute, and these instructions take the form of a bunch of ‘statements’. Here are some of the rules on statements; you will learn them in more detail as you go through this book.

- A program contains statements, each terminated by a semicolon.
- ‘Curly braces’ can enclose multiple statements.
- A statement can correspond to some action when the program is executed.
- Some statements are definitions, of data or of possible actions.
- Comments are ‘Note to self’, short:

```
cout << "Hello world" << '\n'; // say hi!
```

and arbitrary:

```
cout << /* we are now going  
          to say hello  
*/ "Hello!" << /* with newline: */ '\n' ;
```

In the examples so far you see output statements terminated as:

```
cout << something << "\n";
```

where the ‘backslash-n’ stands for a *newline*.

**Exercise 4.3.** Remove the newline from your print statement(s). Compile and run. What do you observe?

Sometimes you will also see:

```
// at the top of the program:  
using std::endl;
```

```
// among the statements:  
cout << something << endl;
```

which has the same behavior of issuing a newline. The distinction will not be important to you for now; see the discussion in section 12.7 if you’re curious.

**Exercise 4.4.** Take the ‘hello world’ program you wrote above, and duplicate the hello-line. Compile and run.

Does it make a difference whether you have the two hellos on the same line of your file, or on different lines?

Experiment with other changes to the layout of your source. Find at least one change that leads to a compiler error. Can you relate the message to the error?

Your program source can have several types of errors. Distinguishing by when you notice them, we roughly distinguish them as follows. (For details on error handling, see chapter 23.)

1. *Syntax* or *compile-time* errors: these arise if what you write is not according to the language specification. The compiler catches these errors, and it refuses to produce a *binary file*.
2. *Run-time* errors: these arise if your code is syntactically correct, and the compiler has produced an executable, but the program does not behave the way you intended or foresaw. Examples are divide-by-zero or indexing outside the bounds of an array.
3. Design errors: your program does not do what you think it does.

**Review 4.2.** True or false?

- If your program compiles correctly, it is correct.
- If you run your program and you get the right output, it is correct.

In the program you just wrote, the string you displayed was completely up to you. Other elements, such as the `cout` keyword, are fixed parts of the language. Most programs contains them.

You see that certain parts of your program are inviolable:

- There are *keywords* such as `return` or `cout`; you can not change their definition.
- Curly braces and parentheses need to be matched.

- There has to be a `main` keyword.
- The `iostream` and `std` are usually needed.

### 4.2.1 Language vs library and about: using

The examples above had a line

```
#include <iostream>
```

which allowed you to write `cout` in your program for output. The `iostream` is a *header*, and it adds *standard library* functionality to the base language.

Functionality such as `cout` can be used in various ways:

You can spell it out as `std::cout`:

```
#include <iostream>
int main() {
    std::cout "hello\n";
    return 0;
}
```

You can add a `using` statement:

```
#include <iostream>
using std::cout;
int main() {
    cout "hello\n";
    return 0;
}
```

Instead of having separate `using` statements for each library function, you could also use a single line

```
using namespace std;
```

in your program. While it is common to find this in examples online, it is frowned upon; see section 20.4 for a discussion.

**Exercise 4.5.** Experiment with the `cout` statement. Replace the string by a number or a mathematical expression. Can you guess how to print more than one thing, for instance:

- the string *One third is*, and
- the result of the computation  $1/3$ ,

with the same `cout` statement? Do you get anything unexpected?

## 4.3 Variables

A program could not do much without storing data: input data, temporary data for intermediate results, and final results. Data is stored in *variables*, which have

- a name, so that you can refer to them,
- a *datatype*, and
- a value.

Think of a variable as a labeled placed in memory.

- The variable is defined in a *variable declaration*,
- which can include an *variable initialization*.
- After a variable is defined, and given a value, it can be used,
- or given a (new) value in a *variable assignment*.

```
int i, j; // declaration
i = 5;    // set a value
i = 6;    // set a new value
j = i+1; // use the value of i
i = 8;    // change the value of i
           // but this doesn't affect j:
           // it is still 7.
```

### 4.3.1 Variable declarations

A variable is defined, in a *variable declaration*. This associates its name and its type, and possibly an initial value; see section 4.3.2.

Let's first talk about what a variable name can be.

- A variable name has to start with a letter;
- a name can contains letters and digits, but not most special characters, except for the underscore.
- For letters it matters whether you use upper or lowercase: the language is *case sensitive*.
- Words such as `main` or `return` are *reserved words*.
- Usually `i` and `j` are not the best variable names: use `row` and `column`, or other meaningful names, instead.
- While you can start a name with an underscore, there are some limitations on the use of the underscore: do not use two underscores in a row, and do not start a name with an underscore followed by a capital letter.

Next, a *variable declaration* states the type and the name of a variable. If you have multiple variables of the same type, you can combine the declarations.

A variable declaration establishes the name and the type of a variable:

```
int n_elements;
float value;
int row, col;
double re_part, im_part;
```

You can not redeclare a variable like this:

```
int value=5;
cout << value << '\n';
float value=1.3;
cout << value << '\n';
```

but the rules for what **is** allowed are a little harder to state. You'll see that later in chapter 8.

Declarations can go pretty much anywhere in your program, but they need to come before the first use of the variable.

Note: it is legal to define a variable before the main program but such *global variables* are usually not a

good idea. Please only declare variables *inside* main (or inside a function et cetera).

### 4.3.2 Initialization

It is possible to give a variable a value right when it's created. This is known as *initialization* and it's different from creating the variable and later assigning to it (section 4.3.3).

There are (at least) two ways of initializing a variable

```
int i = 5;  
int j{6};
```

Note that writing

```
int i;  
i = 7;
```

is not an initialization: it's a declaration followed by an assignment.

If you declare a variable but not initialize, you can not count on its value being anything, in particular not zero. While some compilers do this (some of the time), such implicit initialization is also often omitted for performance reasons.

### 4.3.3 Assignments

Setting a variable

```
i = 5;
```

means storing a value in the memory location. It is not the same as defining a mathematical equality

let  $i = 5$ .

Once you have declared a variable, you need to establish a value. This is done in an *assignment* statement. After the above declarations, the following are legitimate assignments:

```
n = 3;  
x = 1.5 - n;  
n1 = 7;  
n2 = n1 * 3;
```

These are not math equations: the variable on the left hand side gets the value of the expression on the right hand side.

You see that you can assign both a simple value or an expression.

You can set the value of a variable multiple times.

```
int i;  
i = 5;  
// do something with i  
i = 6;  
// do something with i
```

You can also update the value of a variable, using its current value:

```
i = 2*i + 1;
```

Certain assignments with the same variable in both the left and right hand sides can be simplified:

```
x = x+2; y = y/3;
// can be written as
x += 2; y /= 3;
```

Integer add/subtract one:

```
i=i+1; j=j-1;
// rewritten as:
++i; --j;
// or
i++; j--;
```

There are various levels of programming errors. The following program uses the variable `i` without having given it a value.

#### Exercise 4.6.

```
#include <iostream>
using std::cout;
int main() {
    int i;
    int j = i+1;
    cout << j << "\n";
    return 0;
}
```

What happens?

1. Compiler error
2. Output: 1
3. Output is undefined
4. Error message during running the program.

#### 4.3.4 Datatypes

You have seen a couple of datatypes that variables can have. We'll go into the issue of datatypes into a little more detail.

Variables come in different types;

- We call a variable of type `int`, `float`, `double` a *numerical variable*.
- *Complex numbers* will be discussed later.
- For characters: `char`. Strings are complicated; see later.
- Truth values: `bool`
- You can make your own types. Later.

For complex numbers see section 24.1. For strings see chapter 11.

#### 4.3.4.1 Integers

Mathematically, integers are a special case of real numbers. In a computer, integers are stored very differently from real numbers – or technically, floating point numbers.

You might think that C++ integers are stored as binary number with a sign bit, but the truth is more subtle. For now, know that within a certain range, approximately symmetric around zero, all integer values can be represented.

**Exercise 4.7.** These days, the default amount of storage for an `int` is 32 bits. After one bit is used for the sign, that leave 31 bits for the digits. What is the representable integer range?

The integer type in C++ is `int`:

```
int my_integer;
my_integer = 5;
cout << my_integer << "\n";
```

For more integer types, see section 26.3.2; if you’re wondering how large integers and other numeric types can get numerically, see section 24.2.

Integer constants can be represented in several bases.

Integers are normally written in decimal, and stored in 32 bits. If you need something else:

```
int d = 42;           // decimal
int o = 052;          // octal: start with zero
int x = 0x2a;          // hex
int X = 0X2A;          // also hex
int b = 0b101010; // binary
long ell = 42L;
```

Binary numbers are new to C++17.

#### 4.3.4.2 Floating point numbers

*Floating point number* is the computer science-y name for scientific notation: a number written like

$$+6 \cdot 022 \times 10^{23}$$

with:

- an optional sign;
- an integral part;
- a decimal point, or more generally *radix point* in other number bases;
- a fractional part, also known as *mantissa* or *significand*;
- and an *exponent part*: base to some power.

Floating point numbers are by default of type `double`, which standard for ‘double precision’. Double of what? We will discuss that in section 24.2. For now, let’s discuss only the matter of how they are represented.

Without further specification, a floating point literal is of type `double`:

```
1.5
1.5e+5
```

Use a suffix `1.5f` for type `float` which stands for ‘single precision’:

```
1.5f
1.5e+5f
```

Use a suffix `1.5L` for `long double`.

```
1.5L
1.5e+5L
```

A hexadecimal representation of floating points number is possible, but somewhat tricky and counterintuitive.

**4.3.4.2.1 Storage sizes** C++ floating point data types correspond to certain formats defined in the IEEE 754 standard (section HPC book [13], section 3.4). The most common floating point types are:

- `float`, the IEEE 32-bit single precision type,
- `double`, the IEEE 64-bit double precision type,
- `long double`, a compiler / architecture dependent type that is at least as precise as double precision.

There are two 16-bit floating point types. The `float16_t` is the type originally defined in the IEEE 754 standard; the `bfloat16_t` has the layout that you get from the IEEE 32-bit type by omitting the last two bytes. This makes sense in Machine Learning (ML) applications, both from a precision point of view, and for increased bandwidth over using 32-bit floats.

**Remark** *The half precision type makes sense when less accuracy is required; conversely, the quadruple precision type is needed in some numerical applications. However, while these types may be available in a language implementation, this can be without hardware support, which would make their use very slow.*

**4.3.4.2.2 Limitations** Floating point numbers are also referred to as ‘real numbers’ (in fact, in the Fortran language they are defined with the keyword `Real`), but this is sloppy wording. Since only a finite number of bits/digits is available, only terminating fractions are representable. For instance, since computer numbers are binary,  $1/2$  is representable but  $1/3$  is not.

**Exercise 4.8.** Can you think of a way that non-terminating fractions, including such numbers such as  $\sqrt{2}$ , would still be representable?

- You can assign variables of one type to another, but this may lead to truncation (assigning a floating point number to an integer) or unexpected bits (assigning a single precision floating point number do a double precision).

Floating points numbers do not behave like mathematical numbers; for extensive discussion, see HPC book [13], section 3.3 and later.

Floating point arithmetic is full of pitfalls.

- Don't count on  $3 * (1./3)$  being exactly 1.
- Not even associative.

Complex numbers exist, see section 24.1.

**4.3.4.2.3 Don't believe everything you see** You may display a number with `cout` and believe that what's on the screen is what's in the computer. That's not necessarily true, because the number that's stored is formatted, for instance with a default number of digits.

Here is an example. We compute  $\pi$  as a `float` and `double`; they display the same, but computing the difference we see that they are not the same:

**Code:**

```
// basic/pi.cpp
float pi_f = std::atan(1.f)*4.f;
double pi_d = std::atan(1.)*4.;
cout << "Float pi="
    << pi_f << '\n';
cout << "Double pi="
    << pi_d << '\n';
cout << "Difference (d - f) : "
    << "\n"
    << pi_d-pi_f << '\n';
```

**Output:**  
**[basic] pipi:**

```
Float pi=3.14159
Double pi=3.14159
Difference (d - f) :
-8.74228e-08
```

#### 4.3.4.3 Boolean values

So far you have seen integer and real variables. There are also *boolean values* which represent truth values. There are only two values: `true` and `false`.

```
bool found{false};
found = true;
```

**Remark** Sometimes it's convenient to use integers to represent truth values: zero is false, and anything nonzero is true. Don't make a habit of this: if your algorithm needs a truth value, use the `bool` type.

## 4.4 Input/Output, or I/O as we say

A program typically produces output. For now we will only display output on the screen, but output to file is possible too. Regarding input, sometimes a program has all information for its computations, but it is also possible to base the computation on user input.

Terminal (console) output with `cout`:

```
float x = 5;
cout << "Here is the root: " << sqrt(x) << '\n';
```

Note the newline character.

Alternatively: `std::endl`, less efficient.

You can get input from the keyboard with `cin`, which accepts arbitrary strings, as long they don't have spaces.

```
// basic/cin.cpp
string name; int age;
cout << "Your name?\n";
cin >> name;
cout << "age?\n";
cin >> age;
cout << age << " is a nice age, "
     << name << '\n';

> ./cin
Your name?
Victor
age?
18
18 is a nice age, Victor
> ./cin
Your name?
THX 1138
age?
1138 is a nice age, THX
```

For more flexible input, see section 12.8.

For fine-grained control over the output, see section 12.4.1. For other I/O related matters, such as file I/O, see chapter 12.

## 4.5 Expressions

The most basic step in computing is to form expressions such as sums, products, logical conjunctions, string concatenations from variables and constants.

Let's start by discussing constants: numbers, truth values, strings.

### 4.5.1 Numerical expressions

Expressions in programming languages for the most part look the way you would expect them to.

- Mathematical operators: `+` `-` `/` and `*` for multiplication.
- Integer modulus: `5\char`\%2`
- You can use parentheses: `5*(x+y)`. Use parentheses if you're not sure about the precedence rules for operators.
- C++ does not have a power operator (Fortran does): 'Power' and various mathematical functions are realized through library calls.

Math functions are in `cmath`:

```
#include <cmath>
.....
x = pow(3,.5);
```

For squaring, usually better to write `x*x` than `pow(x, 2)`.

**Exercise 4.9.** Write a program that :

- displays the message `Type a number,`
- accepts an integer number from you (use `cin`),
- makes another variable that is three times that integer plus one,
- and then prints out the second variable.

Fine points of integers and integer expression are discussed in section 26.3.2.

### 4.5.2 Truth values

In addition to numerical types, there are truth values, `true` and `false`, with all the usual logical operators defined on them.

- Relational operators: `== != < > <= >=`
- Boolean operators: `not, and, or` (oldstyle: `! && ||`);

### 4.5.3 Type conversions

Since a variable has one type, and will always be of that type, you may wonder what happens with

```
float x = 1.5;  
int i;  
i = x;
```

or

```
int i = 6;  
float x;  
x = i;
```

- Assigning a floating point value to an integer truncates the latter.
- Assigning an integer to a floating point variable fills it up with zeros after the decimal point.

**Exercise 4.10.** Try out the following:

- What happens when you assign a positive floating point value to an integer variable?  
What happens when you assign a negative floating point value to an integer variable?  
Does your compiler give warnings? Is there a way you can trick the compiler into not understanding what you are doing?
- What happens when you assign a `float` to a `double`? Try various numbers for the original float. Print out the result, and if they look the same, see if the difference is actually zero.

The rules for type conversion in expressions are not entirely logical. Consider

```
float x; int i=5, j=2;  
x = i/j;
```

This will give 2 and not 2.5, because `i/j` is an integer expression and is therefore completely evaluated as such, giving 2 after truncation. The fact that it is ultimately assigned to a floating point variable does not cause it to be evaluated as a computation on floats.

You can force the expression to be computed in floating point numbers by writing

```
x = (1.*i)/j;
```

or any other mechanism that forces a conversion, without changing the result. Another mechanism is the *cast*; this will be discussed in section 26.2.

**Exercise 4.11.** Write a program that asks for two integer numbers  $n_1, n_2$ .

- Assign the integer ratio  $n_1/n_2$  to an integer variable.
- Can you use this variable to compute the modulus

$$n_1 \bmod n_2$$

(without using the `\char`%` modulus operator!)

Print out the value you get.

- Also print out the result from using the modulus operator; `\char`%`.
- Investigate the behavior of your program for negative inputs. Do you get what you were expecting?

**Exercise 4.12.** Write two programs, one that reads a temperature in Centigrade and converts to Fahrenheit, and one that does the opposite conversion.

$$C = (F - 32) \cdot 5/9, \quad F = 9/5 C + 32$$

Check your program for the freezing and boiling point of water.

(Do you know the temperature where Celsius and Fahrenheit are the same?)

Can you use Unix pipes to make one accept the output of the other?

**Review 4.3.** True or false?

1. Within a certain range, all integers are available as values of an integer variable.
2. Within a certain range, all real numbers are available as values of a float variable.
3.  $5(7+2)$  is equivalent to 45.
4.  $1--1$  is equivalent to zero.
5. `int i = 5/3.;` The variable `i` is 2.
6. `float x = 2/3;` The variable `x` is approximately 0.6667.

## 4.5.4 Characters and strings

In this course we are mostly concerned with numerical data, but string and character data can be useful for purposes of output.

### 4.5.4.1 Strings

Strings, that is, strings of characters, are not a C++ built-in datatype. Thus, they take some extra setup to use. See chapter 11 for a full discussion.

The `char` data type is used for single characters, and `string` is used for strings.

- Add the following at the top of your file:

```
#include <string>
using std::string;
```

- Declare string variables as

```
string name;
```

- And you can now `cin` and `cout` them.

A character is enclosed in single quotes:

'x'

while a general string is enclosed in double quotes:

"The quick brown fox"

**Exercise 4.13.** Write a program that asks for the user's first name, uses `cin` to read that, and prints something like *Hello, Susan!* in response.

What happens if you enter first and last name?

## 4.6 Advanced topics

### 4.6.1 The main program and the return statement

The `main` program has to be of type `int`; however, many compilers tolerate deviations from this, for instance accepting `void`, which is not language standard.

The arguments to `main` can be:

```
int main()
int main( int argc, char* argv[] )
int main( int argc, char **argv )
```

The `argc/argv` variables contain the commandline as a set of strings.

- `argc` is the number of strings: the name of the program, and the number of space-separated arguments;
- `argv` contains the *commandline arguments* as an array of strings.

You might be tempted to parse the commandline yourself, but there are dedicated libraries for this; see section 63.2.2.

The returned `int` can be specified several ways:

- If no `return` statement is given, implicitly `return 0` is performed.
- If you explicitly use `return` with an integer value.
- Instead of an explicit integer value you can use the values `EXIT_SUCCESS` and `EXIT_FAILURE` which are defined in `cstdlib`. In general, zero indicates success, while a nonzero value indicates failure.
- You can also use the `exit` function:

```
void exit(int);
```

The point of having a return code is that it is passed to the operating system as a *return code*, which can then be queried in the *shell*.

<b>Code:</b> <pre><code>// basic/return.cpp int main() {     return 1; }</code></pre>	<b>Output:</b> <pre><code>[basic] return: ./return ; \     if [ \$? -ne 0 ] ; then \         echo "Program failed" ; \     fi Program failed</code></pre>
--	--

## 4.6.2 Identifier names

Variable names, or more correctly: *identifiers*, have to start with a non-digit. To be precise, this can be

- a Latin letter, which is the most common case;
- an *underscore*, which is the convention for private members of a class, and other ‘internal’ names; or
- *Unicode* characters of class *XID\\_Start*.

Any following character can be *Unicode* characters of class *XID\\_Continue*.

On the topic of underscores, a leading *double underscore* should not be used since such names are reserved for the compiler.

**Remark** General *Unicode* characters became allowed in C++23, but this convention was then applied retro-actively to earlier standards.

## 4.7 C differences

### 4.7.1 Boolean

Traditionally, C did not have a type for boolean values; instead `int` and `short` was used, where zero was false, and any nonzero value true. In C99 the type `_Bool` was introduced. This only serves legibility: there are no true/false constants, and variables of type `_Bool` still have to be treated as integers in `printf`.

```
// c/ctypes.c
_Bool tf = 1;
printf("True: %d\n", tf);
```

However, the `stdbool.h` defines `bool`, `true`, and `false` as aliases.

## 4.8 Review questions

#### 4. Basic elements of C++

---

**Review 4.4.** Name the elements of the following commandline and their function:

```
icpx -o myprogram myprogram.cpp
```

**Review 4.5.** What is the output of:

```
int m=32, n=17;  
cout << n%m << "\n";
```

**Review 4.6.** Given

```
int n;
```

give an expression that uses elementary mathematical operators to compute  $n^3$ . Do you get the correct result for all  $n$ ? Explain.

How many elementary operations does the computer perform to compute this result?

Can you now compute  $n^6$ , minimizing the number of operations the computer performs?

# Chapter 5

## Conditionals

A program consisting of just a list of assignment and expressions would not be terribly versatile. At least you want to be able to say ‘if some condition, do one computation, otherwise compute something else’, or: ‘until some test is true, iterate the following computations’. The mechanism for testing and choosing an action accordingly is called a *conditional*. (Iterating is discussed in chapter 6.)

### 5.1 Conditionals

Here are some forms a conditional can take.

A single statement, executed if the test is true:

```
if (x<0)
    x = -x;
```

Single statement in the true branch, and likewise a single statement in the false branch:

```
if (x>=0)
    x = 1;
else
    x = -1;
```

Both in the true and the false branch multiple statements are allowed, if you enclose them in curly braces:

```
if (x<0) {
    x = 2*x; y = y/2;
} else {
    x = 3*x; y = y/3;
}
```

You can chain conditionals by extending the `else` part. In this example the dots stand for omitted code:

```
if (x>0) {
    ....
} else if (x<0) {
    ....
} else {
    ....
}
```

Conditionals can also be nested:

## 5. Conditionals

---

```
if (x>0) {  
    if (y>0) {  
        ....  
    } else {  
        ....  
    }  
} else {  
    ....  
}
```

- In that last example the outer curly brackets in the true branch are optional. But it's safer to use them anyway.
- When you start nesting constructs, use indentation to make it clear which line is at which level. A good editor helps you with that.

**Exercise 5.1.** For what values of  $x$  will the left code print 'b'?

For what values of  $x$  will the right code print 'b'?

<pre>float x = /* something */ if ( x &gt; 1 ) {     cout &lt;&lt; "a" &lt;&lt; endl;     if ( x &gt; 2 )         cout &lt;&lt; "b" &lt;&lt; endl; }</pre>	<pre>float x = /* something */ if ( x &gt; 1 ) {     cout &lt;&lt; "a" &lt;&lt; endl; } else if ( x &gt; 2 ) {     cout &lt;&lt; "b" &lt;&lt; endl; }</pre>
--	---

## 5.2 Operators

You have already seen arithmetic expressions; now we need to look at logical expressions: just what can be tested in a conditional. For the most part, logical expressions are intuitive. However, note that they can be chained only in certain ways:

```
bool x,y,z;  
if ( x or y or z ) ; //good  
int i,j,k;  
if ( i < j < k ) ; // WRONG
```

Here are the most common *logic operators* and *comparison operators*:

Operator	meaning	example
$==$	equals	$x == y - 1$
$\neq$	not equals	$x * x \neq 5$
$>$	greater	$y > x - 1$
$\geq$	greater or equal	$\text{sqrt}(y) \geq 7$
$<, \leq$	less, less equal	
$\&, \mid \mid$	and, or	$x < 1 \ \&\ x > 0$
and,or	and, or	$x < 1 \ \text{and} \ x > 0$
!	not	$\text{!} ( x > 1 \ \&\ x < 2 )$
not		$\text{not} ( x > 1 \ \text{and} \ x < 2 )$

Precedence rules of operators are common sense. When in doubt, use parentheses.

**Exercise 5.2.** The following code claims to detect if an integer has more than 2 digits.

**Code:**

```
// basic/if.cpp
int i;
cin >> i;
if ( i>100 )
    cout << "That number " << i
        << " had more than 2 digits"
        << '\n';
```

**Output:**

```
[basic] if:
... with 50 as input ....
... with 150 as input ....
That number 150 had more than 2
→digits
```

Fix the small error in this code. Also add an ‘else’ part that prints if a number is negative.

You can base this off the file `if.cpp` in the repository

**Exercise 5.3.** Read in an integer. If it is even, print ‘even’, otherwise print ‘odd’:

```
if ( /* your test here */ )
    cout << "even" << '\n';
else
    cout << "odd" << '\n';
```

Then, rewrite your test so that the true branch corresponds to the odd case.

In exercise 5.3 it didn’t matter whether you used the test for even or for odd, but sometimes it does make a difference how you arrange complicated conditionals. In the following exercise, think about how to arrange the tests, iff possible in more than one way.

**Exercise 5.4.** Read in a positive integer. If it’s a multiple of three print ‘Fizz!'; if it’s a multiple of five print ‘Buzz!'. It is a multiple of both three and five print ‘Fizzbuzz!'. Otherwise print nothing.

Note:

- Capitalization.
- Exclamation mark.
- Your program should display at most one line of output.

## 5.2.1 Bitwise logic

Above we only considered the `and` and `or` logical operators, also spelled `&&` and `||`. There are also *bitwise operators*, that look confusingly similar to the latter notation, but that operate on each bit of their operands independently.

## 5. Conditionals

**Code:**

```
// basic/bitor.cpp
int x=6, y=3;
cout << "6|3 = " << (x|y)
     << '\n';
cout << "6&3 = " << (x&y)
     << '\n';
```

**Output:**

```
[basic] bitor:
6|3 = 7
6&3 = 2
```

To understand what's happening here, realize that

$$6_{10} \equiv 110_2 \quad \text{and} \quad 3_{10} \equiv 011_2$$

where the subscript indicates the base.

**Exercise 5.5.** How would you test if a number is odd or even with bitwise testing?

How many other ways can you find to test odd/even-ness?

You will probably not use the bitwise operators often, but the following idiom is sometimes encountered:

```
const int
    STATE_1 = 1, STATE_2 = 1<<1, STATE_3 = 1<<2;
int state = /* stuff */;
if ( state & ( STATE_1 | STATE_3 ) )
    cout << "We are in state 1 or 3";
```

### 5.3 Switch statement

If you have a number of cases corresponding to specific integer values, you can use the `switch` statement. While anything you can code with the `switch` statement can also be coded with conditionals, the compiler may generate more efficient code with `switch`.

Cases are evaluated consecutively until you ‘break’ out of the switch statement:

```
Code:
// basic/switch.cpp
switch (n) {
case 1 :
case 2 :
    cout << "very small" << '\n';
    break;
case 3 :
    cout << "trinity" << '\n';
    break;
default :
    cout << "large" << '\n';
}
```

```
Output:
[basic] switch:
for v in 1 2 3 4 5 ; do \
    echo $v | ./switch ; \
done
very small
very small
trinity
large
large
```

**Exercise 5.6.** Suppose the variable *n* is a nonnegative integer. Write a `switch` statement that has the same effect as:

```
if (n<5)
    cout << "Small" << endl;
else
    cout << "Not small" << endl;
```

## 5.4 Scopes

The true and false branch of a conditional can consist of a single statement, or of a block in curly brackets. Such a block creates a **scope** where you can define local variables.

```
if ( something ) {
    int i;
    .... do something with i
}
// the variable 'i' has gone away.
```

See chapter 8 for more on scopes.

## 5.5 Advanced topics

### 5.5.1 Short-circuit evaluation

C++ logic operators have a feature called *short-circuit evaluation*: a logical operator stops evaluating in strict left-to-right order when the result is clear. For instance, in

`clause1 and clause2`

the second clause is not evaluated if the first one is `false`, because the truth value of this conjunction is already determined.

Likewise, in

`clause1 or clause2`

## 5. Conditionals

---

the second clause is not evaluated if the first one is `true`, because the value of the `or` conjunction is already clear.

This mechanism allows you to write

```
if ( x>=0 and sqrt(x)<10 ) { /* ... */ }
```

Without short-circuit evaluation the square root operator could be applied to negative numbers.

### 5.5.2 Ternary if

The true and false branch of a conditional contain whole statements. For example

```
if (foo)
    x = 5;
else
    y = 6;
```

But what about the case where the true and false branch assign to the same variable, but with a different expression? You can not write

Original code:

```
if (foo)
    x = 5;
else
    x = 6;
```

Not legal syntax for ‘simplification’:

```
x = if (foo) 5; else 6;
```

For this case, the *ternary if* acts as if it’s an expression itself, but chosen between two expressions. The previous assignment to `x` then becomes:

```
x = foo ? 5 : 6;
```

Surprisingly, this expression can even be in the left-hand side:

```
foo ? x : y = 7;
```

### 5.5.3 Initializer

The C++17 standard introduced a new form of the `if` and `switch` statement: it is possible to have a single statement of declaration prior to the test. This is called the *initializer*.

Variable local to the conditional:
------------------------------------

**Code:**

```
// basic/ifinit.cpp
if ( char c = getchar(); c != 'a' )
    cout << "Not an a, but: "
    << c << '\n';
else
    cout << "That was an a!"
    << '\n';
```

**Output:**

```
[basic] ifinit:
for c in d b a z ; do \
    echo $c | ./ifinit ; \
done
Not an a, but: d
Not an a, but: b
That was an a!
Not an a, but: z
```

This is particularly elegant if the init statement is a declaration, because the declared variable is then local to the conditional. Previously one would have had to write

```
char c;
c = getchar();
if ( c != 'a' ) /* ... */
```

with the variable defined outside of the scope of the conditional.

You can have further initializers in `else if` branches. The scope of variables defined there spans any textually following branches.

If you already know how to write functions, you can do the following exercises.

**Exercise 5.7.** Write a function `float read_number()` and use it in a conditional.

```
if ( /* something */ ) {
    cout << "The root of this non-negative number is: "
    << sqrt(number) << '\n';
} else
    cout << "Number was negative\n";
```

Make sure to use an initializer; `number` should be limited in scope to the conditional.

**Exercise 5.8.** Write a function `int read_number()` and use it to rewrite your fizzbuzz solution.

Make sure to use an initializer; the number you are testing should be limited in scope to the conditional.

## 5.6 Review questions

**Review 5.1.** True or false?

1. The tests `if (i>0)` and `if (0<i)` are equivalent.
2. The test

```
if (i<0 && i>1)
    cout << "foo"
```

prints `foo` if  $i < 0$  and also if  $i > 1$ .

3. The test

```
if (0<i<1)
```

## 5. Conditionals

---

```
cout << "foo"
```

prints `foo` if  $i$  is between zero and one.

**Review 5.2.** Do you think this is good code?

```
bool x;  
// ... code that sets x ...  
if ( x == true )  
    // do something
```

**Review 5.3.** T/F: the following is a legal program:

```
#include <iostream>  
int main() {  
    if (true)  
        int i = 1;  
    else  
        int i = 2;  
    std::cout << i;  
    return 0;  
}
```

**Review 5.4.** T/F: the following are equivalent:

```
if (cond1)  
    i = 1;  
else if (cond2)  
    i = 2;  
else  
    i = 3;
```

compare:

```
if (cond1)  
    i = 1;  
else {  
    if (cond2)  
        i = 2;  
    else  
        i = 3;  
}
```

**Review 5.5.** Find at least one error in this program

```
int score; // assume that this has a value 1–100  
string grade;  
if (score>92)  
    grade = "A";  
if (score<92 and score>85)  
    grade = "A-";  
if (score<85 and score>78)  
    grade = "B+";  
// a couple more cases follow here  
if (score<30)  
    grade = "F";
```

Critique the basic design of this code.

# Chapter 6

## Looping

There are many cases where you want to repeat an operation or series of operations:

- A time-dependent numerical simulation executes a fixed number of steps, or until some stopping test.
- Recurrences:

$$x_{i+1} = f(x_i).$$

- Inspect or change every element of a database table.

The C++ construct for such repetitions is known as a *loop*: a number of statements that get repeated. The two types of loop statement in C++ are:

- the *for loop* which is typically associated with a pre-determined number of repetitions, and with the repeated statements being based on a counter of some sort; and
- the *while loop*, where the statements are repeated indefinitely until some condition is satisfied.

However, the difference between the two is not clear-cut: in many cases you can use either.

We will now first consider the `for` loop; the `while` loop comes in section 6.3.

### 6.1 The ‘for’ loop

In the most common case, a for loop has a *loop counter*, ranging from some initial value to some final value. An example showing the syntax for this simple case is:

```
int sum_of_squares{0};
for (int var=low; var<upper; var++) {
    sum_of_squares += var*var;
}
cout << "The sum of squares from "
     << low << " to " << upper
     << " is " << sum_of_squares << endl;
```

The `for` line is called the *loop header*, and the statements between curly brackets the *loop body*. Each execution of the loop body is called an *iteration*.

## 6. Looping

---

**Exercise 6.1.** Read an integer value with `cin`, and print ‘Hello world’ that many times.

**Exercise 6.2.** Extend exercise 6.1: the input 17 should now give lines

```
Hello world 1  
Hello world 2  
....  
Hello world 17
```

Can you do this both with the loop index starting at 0 and 1?

Also, let the numbers count down.

We will now investigate the components of a loop.

### 6.1.1 Loop variable

First of all, most `for` loops have a *loop variable* or *loop index*. The first expression in the parentheses is usually concerned with the initialization of this variable: it is executed once, before the loop iterations. If you declare a variable here, it becomes local to the loop, that is, it only exists in the expressions of the loop header, and during the loop iterations.

The loop variable is usually an integer:

```
for ( int index=0; index<max_index; index=index+1 ) {  
    ...  
}
```

But other types are allowed too:

```
for ( float x=0.0; x<10.0; x+=delta ) {  
    ...  
}
```

Beware the stopping test for non-integral variables!

**Exercise 6.3.** Write a loop that prints  $x = 1/10, 2/10, \dots, 1$ . Do this

- with an integer loop variable
- with a `float` or `double` loop variable

How do you do the stopping test?

What do you observe?

Usually, the loop variable only has meaning inside the loop so it should only be defined there. You do this by defining it in the loop header:

```
for (int var=low; var<upper; var++) {
```

However, it can also be defined outside the loop:

```
int var;  
for (var=low; var<upper; var++) {
```

but you should only do this if the variable is actually needed after the loop. You will see an example where this makes sense below in section 6.3.

### 6.1.2 Stopping test

Next there is a test, which needs to evaluate to a boolean expression. This test is often called a ‘stopping test’, but to be technically correct it is actually executed at the start of each iteration, including the first one, and it is really a ‘loop while this is true’ test.

```
Code:
// basic/pretest.cpp
cout << "before the loop" << '\n';
for (int i=5; i<4; ++i)
    cout << "in iteration "
        << i << '\n';
cout << "after the loop" << '\n';
```

```
Output:
[basic] pretest:
before the loop
after the loop
```

- If this boolean expression is true, do the next iteration.
- Done before the first iteration too!
- Test can be empty. This means no test is applied.

```
for ( int i=0; i<N; i++) { ... }
for ( int i=0; ; i++ ) { ... }
```

Usually, the combination of the initialization and the stopping test determines how many iterations are executed. If you want to perform  $N$  iterations you can write

```
for (int iter=0; iter<N; iter++)
```

or

```
for (int iter=1; iter<=N; iter++)
```

The former is slightly more idiomatic to C++, but you should write whatever best fits the problem you are coding.

The stopping test doesn’t need to be an upper bound. Here is an example of a loop that counts down to a lower bound.

```
for (int var=high; var>=low; var--) { ... }
```

The stopping test can be omitted

```
for (int var=low; ; var++) { ... }
```

if the loops ends in some other way. You’ll see this later.

### 6.1.3 Increment

Finally, after each iteration we need to update the loop variable. Since this is typically adding one to the variable we can informally refer to this as the ‘increment’, but it can be a more general update.

## 6. Looping

---

Increment performed after each iteration. Most common:

- `i++` for a loop that counts forward;
- `i--` for a loop that counts backward;

Others:

- `i+=2` to cover only odd or even numbers, depending on where you started;
- `i*=10` to cover powers of ten.

Even optional:

```
for (int i=0; i<N; ) {  
    // stuff  
    if ( something ) i+=1; else i+=2;  
}
```

This is how a loop is executed.

- The initialization is performed.
- At the start of each iteration, including the very first, the stopping test is performed. If the test is true, the iteration is performed with the current value of the loop variable(s).
- At the end of each iteration, the increment is performed.

C difference: Loop variable in C99. Declaring the loop variable in the loop header is also a modern addition to the C language. Use compiler flag `-std=c99`.

**Exercise 6.4.** Take this code:

```
// loop/sumsquares.cpp  
int sum_of_squares{0};  
for (int var=low; var<upper; ++var) {  
    sum_of_squares += var*var;  
}  
cout << "The sum of squares from "  
     << low << " to " << upper  
     << " is " << sum_of_squares << '\n';
```

and modify it to sum only the squares of every other number, starting at `low`.

Can you find a way to sum the squares of the even numbers  $\geq low$ ?

**Review 6.1.** For each of the following loop headers, how many times is the body executed? (You can assume that the body does not change the loop variable.)

```
for (int i=3; i<10; i++)  
  
for (int i=3; i<=10; i++)  
  
for (int i=0; i<0; i++)
```

**Review 6.2.** What is the last iteration executed?

```
for (int i=1; i<=2; i=i+2)
```

```
for (int i=1; i<=5; i*=2)

for (int i=0; i<0; i--)

for (int i=5; i>=0; i--)

for (int i=5; i>0; i--)
```

### 6.1.4 Loop body

The loop body can be a single statement:

```
int s{0};
for (int i=0; i<N; i++)
    s += i;
```

or a block:

```
int s{0};
for (int i=0; i<N; i++) {
    int t = i*i;
    s += t;
}
```

If it is a block, it is a scope inside which you can declare local variables.

## 6.2 Nested loops

Quite often, the loop body will contain another loop. For instance, you may want to iterate over all elements in a matrix. Both loops will have their own unique loop variable.

```
for (int row=0; row<m; row++)
    for (int col=0; col<n; col++)
        ...
    
```

This is called *loop nest*; the `row` loop is called the *outer loop* and the `col` loop the *inner loop*.

Traversing an index space (whether that corresponds to an array object or not) by `row, col` is called the *lexicographic ordering*.

**Exercise 6.5.** Write an  $i, j$  loop nest that prints out all pairs with

$$1 \leq i, j \leq 10, \quad j \leq i.$$

Output one line for each `i` value.

Now write an  $i, j$  loop that prints all pairs with

$$1 \leq i, j \leq 10, \quad |i - j| < 2,$$

## 6. Looping

---

again printing one line per  $i$  value. Food for thought: this exercise is definitely easiest with a conditional in the inner loop, but can you do it without?

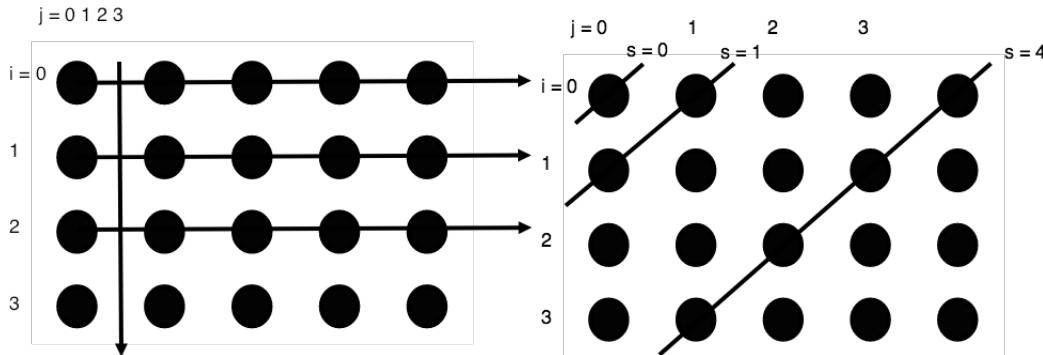


Figure 6.1: Lexicographic and diagonal ordering of an index set

The mere fact that you need to traverse a rectangular range of  $i, j$  indices, does not mean that you have to write a lexicographically indexed loop. Figure 6.1 illustrates that you can look at the  $i, j$  indices by row/column or by diagonal. Just like rows and columns being defined as  $i = \text{constant}$  and  $j = \text{constant}$  respectively, a diagonal is defined by  $i + j = \text{constant}$ .

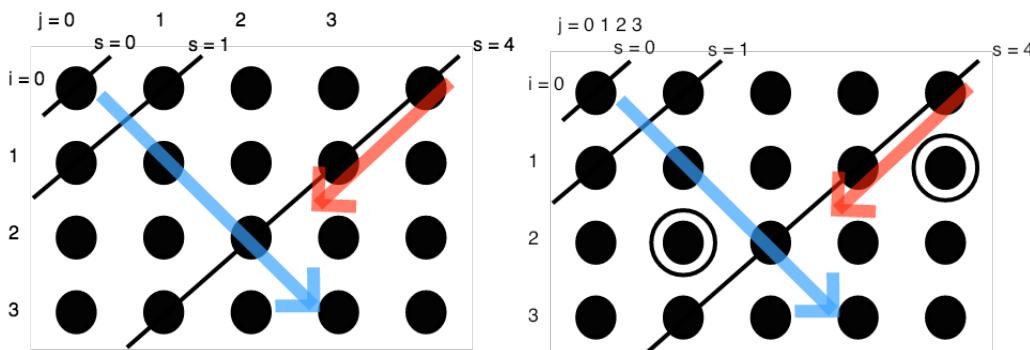


Figure 6.2: Illustration of the second part of exercise 6.7

### 6.3 Looping until

The basic for loop looks pretty deterministic: a loop variable ranges through a more-or-less prescribed set of values. This is appropriate for looping over the elements of an array, but not if you are coding some process that needs to run until some dynamically determined condition is satisfied. In this section you will see some ways of coding such cases.

First of all, the stopping test in the ‘for’ loop is optional, so you can write an indefinite loop as:

```
for (int var=low; ; var=var+1) { ... }
```

How do you end such a loop? For that you use the `break` statement. If the execution encounters this statement, it will continue with the first statement after the loop.

```
for (int var=low; ; var=var+1) {
    // statements;
    if (some_test) break;
    // more statements;
}
```

For the following exercise, see figure 6.2 for inspiration.

**Exercise 6.6.** Write a double loop over  $0 \leq i, j < 10$  that prints all pairs  $(i, j)$  where the product  $i \cdot j > 40$ .

You can base this off the file `ijloop.cpp` in the repository

**Exercise 6.7.** Write a double loop over  $0 \leq i, j < 10$  that prints the first pair where the product of indices satisfies  $i \cdot j > N$ , where  $N$  is a number you read in. A good test case is  $N = 40$ .

Secondly, find a pair with  $i \cdot j > N$ , but with the smallest value for  $i + j$ . (If there is more than one pair, report the one with lower  $i$  value.) Can you traverse the  $i, j$  indices such that they first enumerate all pairs  $i + j = 1$ , then  $i + j = 2$ , then  $i + j = 3$  et cetera? Hint: write a loop over the sum value  $1, 2, 3, \dots$ , then find  $i, j$ .

Your program should print out both pairs, each on a separate line, with the numbers separated with a comma, for instance `8, 5`.

**Exercise 6.8.** All three parts of a loop header are optional. What would be the meaning of

```
for (;;) { /* some code */ }
```

?

Suppose you want to know what the loop variable was when the `break` happened. You need the loop variable to be global:

```
int var;
... code that sets var ...
for ( ; var<upper; var++) {
    ... statements ...
    if (some condition) break
    ... more statements ...
}
... code that uses the breaking value of var ...
```

In other cases: define the loop variable in the header!

Example:

## 6. Looping

---

**Code:**

```
// loop/findmin.cpp
float minpos{0.f};
for ( ; ; minpos+=.5f ) {
    if (f(minpos)>90)
        break;
}
cout << "Minimum satisfying value: "
    << minpos << '\n';
```

**Output:**

```
[loop] findmin:
```

```
Minimum satisfying value: 9.5
Minimum satisfying value: 9.5
```

### Exercise 6.9.

Can you make this loop more compact?

Instead of using a `break` statement, there can be other ways of ending the loop.

If the test comes at the start or end of an iteration, you can move it to the loop header:

```
bool need_to_stop{false};
for (int var=low; !need_to_stop ; var++) {
    ... some code ...
    if ( some condition )
        need_to_stop = true;
}
```

Another mechanism to alter the control flow in a loop is the `continue` statement. If this is encountered, execution skips to the start of the next iteration.

```
for (int var=low; var<N; var++) {
    statement;
    if (some_test) {
        statement;
        statement;
    }
}
```

Alternative:

```
for (int var=low; var<N; var++) {
    statement;
    if (!some_test) continue;
    statement;
    statement;
}
```

The only difference is in layout.

### 6.3.1 While loops

The other possibility for ‘looping until’ is a `while` loop, which repeats until a condition is met. The while loop does not have a counter or an update statement; if you need those, you have to create them yourself.

Syntax:

```
while ( condition ) {
    statements;
}
```

or

```
do {
    statements;
} while ( condition );
```

The two while loop variants can be described as ‘pre-test’ and ‘post-test’. The choice between them entirely depends on context.

```
float money = inheritance();
while ( money < 1.e+6 )
    money += on_year_savings();
```

Let’s consider an example: we read numbers from the input until one is positive. The following two code examples use the `do ... while` and `while ... do` idiom respectively.

The first solution can be termed a ‘pre-test’:

**Code:**

```
// basic/whiledo.cpp
cout << "Enter a positive number: " ;
cin >> invar; cout << '\n';
cout << "You said: " << invar << '\n';
while (invar<=0) {
    cout << "Enter a positive number: " ;
    cin >> invar; cout << '\n';
    cout << "You said: " << invar << '\n';
}
cout << "Your positive number was "
    << invar << '\n';
```

**Output:**

```
[basic] whiledo:
Enter a positive number:
You said: -3
Enter a positive number:
You said: 0
Enter a positive number:
You said: 2
Your positive number was 2
```

Problem: code duplication.

The second one uses a ‘post-test’, and you see that here it solves the problem of code duplication;

## 6. Looping

---

```
Code:  
// basic/dowhile.cpp  
int invar;  
do {  
    cout << "Enter a positive number: " ;  
    cin >> invar; cout << '\n';  
    cout << "You said: " << invar << '\n';  
} while (invar<=0);  
cout << "Your positive number was: "  
    << invar << '\n';
```

```
Output:  
[basic] dowhile:  
Enter a positive number:  
You said: -3  
Enter a positive number:  
You said: 0  
Enter a positive number:  
You said: 2  
Your positive number was: 2
```

The post-test syntax leads to more elegant code.

**Exercise 6.10.** At this point you are ready to do the exercises in the prime numbers project, section 45.3.

**Exercise 6.11.** A horse is tied to a post with a 1 meter elastic band. A spider that was sitting on the post starts walking to the horse over the band, at 1cm/sec. This startles the horse, which runs away at 1m/sec. Assuming that the elastic band is infinitely stretchable, will the spider ever reach the horse?

**Exercise 6.12.** One bank account has 100 dollars and earns a 5 percent per year interest rate. Another account has 200 dollars but earns only 2 percent per year. In both cases the interest is deposited into the account.

After how many years will the amount of money in the first account be more than in the second? Solve this with a `while` loop.

Food for thought: compare solutions with a pre-test and post-test, and also using a for-loop.

## 6.4 Advanced topics

### 6.4.1 Parallelism

At the start of this chapter we mentioned the following examples of loops:

- A time-dependent numerical simulation executes a fixed number of steps, or until some stopping test.
- Recurrences:

$$x_{i+1} = f(x_i).$$

- Inspect or change every element of a database table.

The first two cases actually need to be performed in sequence, while the last one corresponds more to a mathematical ‘forall’ quantor. You will later learn two different syntaxes for this in the context of arrays. This difference can also be exploited when you learn *parallel programming*. Fortran has a *do concurrent* loop construct for this.

## 6.5 Exercises

**Exercise 6.13.** Find all triples of integers  $u, v, w$  under 100 such that  $u^2 + v^2 = w^2$ . Make sure you omit duplicates of solutions you have already found.

**Exercise 6.14.** The integer sequence

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } u_n \text{ is even} \\ 3u_n + 1 & \text{if } u_n \text{ is odd} \end{cases}$$

leads to the *Collatz conjecture*: no matter the starting guess  $u_1$ , the sequence  $n \mapsto u_n$  will always terminate at 1.

5 → 16 → 8 → 4 → 2 → 1

7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 ⋯

(What happens if you keep iterating after reaching 1?)

Try all starting values  $u_1 = 1, \dots, 1000$  to find the values that lead to the longest sequence: every time you find a sequence that is longer than the previous maximum, print out the starting number.

**Exercise 6.15.** Large integers are often printed with a comma (US usage) or a period (European usage) between all triples of digits. Write a program that accepts an integer such as 2542981 from the input, and prints it as 2, 542, 981.

## 6. Looping

---

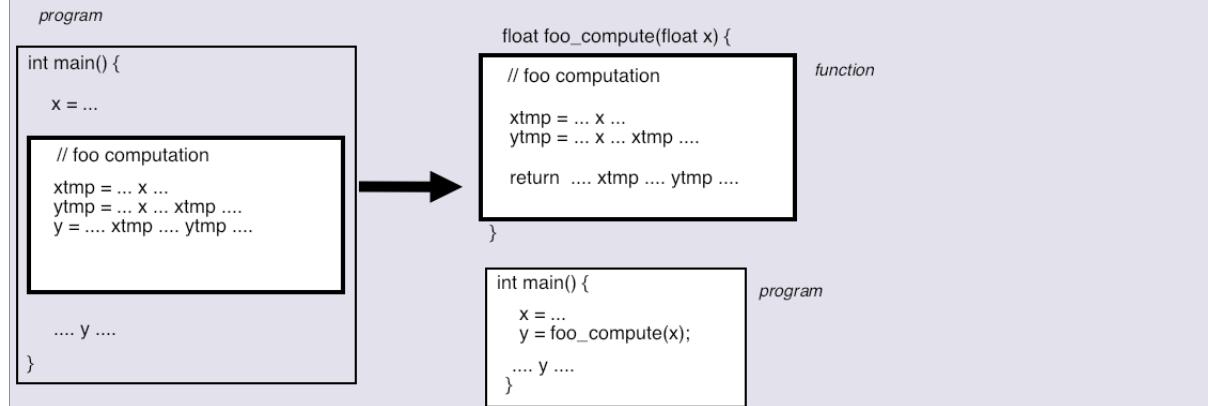
# Chapter 7

## Functions

A *function* (or *subprogram*) is a way to abbreviate a block of code and replace it by a single line. This is foremost a code structuring device: by giving a function a relevant name you introduce the terminology of your application into your program.

- Find a block of code that has a clearly identifiable function.
- Turn this into a function: the function definition will contain that block, with a header that names it.
- The function is called by its name.

Transforming a single file to main and function:



By introducing a function name you have introduced *abstraction*: your program now uses terms related to your problem, and not just the basic control structures such as `for`. With objects (chapter 9) you will learn further abstractions, so that instead of integers and arrays your program will use application terms, such as *Point* or *Line*.

### 7.1 Function definition and call

There are two aspects to a function:

- The *function definition* is done once, typically above the main program;
- a *function call* to any function can happen multiple times, inside the main program or inside other functions.

## 7. Functions

---

Let's consider a simple example program, in which we introduce functions. We code the *bisection* algorithm for finding the root of a function; see section 47.1 for details.

Example: zero-finding through bisection.

$$\underset{x}{?} : f(x) = 0, \quad f(x) = x^3 - x^2 - 1$$

(where the question mark quantor stands for ‘for which  $x$ ’).

First attempt at coding this: everything in the main program.

**Code :**

```
// func/bisect1.cpp
float left{0.}, right{2.},
mid;
while (right-left>.1) {
    mid = (left+right)/2.;
    float fmida =
        mid*mid*mid - mid*mid-1;
    if (fmida<0)
        left = mid;
    else
        right = mid;
}
cout << "Zero happens at: " << mid << '\n';
```

**Output :**

```
[func] bisect1:
Zero happens at: 1.4375
```

We modularize this in two steps. The first function we introduce is the objective function  $f(x)$ .

Introduce a function for the expression  $m*m*m - m*m-1$ :

```
// func/bisect2.cpp
float f(float x) {
    return x*x*x - x*x-1;
};
```

Used in main:

```
// func/bisect2.cpp
while (right-left>.1) {
    mid = (left+right)/2.;
    float fmida = f(mid);
    if (fmida<0)
        left = mid;
    else
        right = mid;
}
```

Next we introduce a function for the zero-finding algorithm.

Function:

```
// func/bisect3.cpp
float f(float x) {
    return x*x*x - x*x-1;
}
float find_zero_between
    (float l,float r) {
    float mid;
    while (r-l>.1) {
        mid = (l+r)/2.;
        float fmid = f(mid);
        if (fmid<0)
            l = mid;
        else
            r = mid;
    }
    return mid;
}
```

New main:

```
// func/bisect3.cpp
int main() {
    float left{0.},right{2.};
    float zero =
        find_zero_between(left,right);
    cout << "Zero happens at: "
        << zero << '\n';
    return 0;
}
```

Simple as this example is, it illustrates the process of isolating code bits into functions. One guideline that sometimes is given is the ‘single-responsibility principle’: each function needs to do one clearly delineated thing. Phrased more conservatively: don’t make your function messy by letting it do too many things.

The main now no longer contains any implementation details, such as local variables, or method used. This makes the main program shorter and more elegant: we have moved the variables for the midpoint inside the function. These are implementation details and should not be in the main program.

In this example, the function definition consists of:

- The keyword **float** indicates that the function returns a **float** result to the calling code.
- The name *find\_zero\_between* is picked by you.
- The parenthetical part (**float** *l*,**float** *r*) is called the ‘parameter list’: it says that the function takes two floats as input. For purposes of the function, the floats will have the names *l*, *r*, regardless any names in the main program.
- The ‘body’ of the function, the code that is going to be executed, is enclosed in curly brackets.
- A ‘return’ statement that transfers a computed result out of the function.

### 7.1.1 Formal definition of a function definition

Formally, a *function definition* consists of:

- *function result type*: you need to indicate the type of the result;
- name: you get to make this up;
- zero or more *function parameters*. These describe how many *function arguments* you need to supply as input to the function. Parameters consist of a type and a name. This makes them look like variable declarations, and that is how they function. Parameters are separated by commas. Then follows the:

- *function body*: the statements that make up the function. The function body is a *scope*: it can have local variables. (You can not nest function definitions.)
- a *return* statement. Which doesn't have to be the last statement, by the way.

While a function definition gives everything there is to know about a function, the *function declaration* only specifies enough to use a function. For instance, it does not specific parameter names and the function body.

Function declaration:

- Function name
- Return type and types of parameters
- qualifiers (see later)

Function definition

- Declaration, plus
- parameter names and function body

The function can then be used in the main program, or in another function.

A function body defines a *scope*: the local variables of the function calculation are invisible to the calling program.

Functions can not be nested: you can not define a function inside the body of another function.

### 7.1.2 Function call

The *function call* consists of

- The name of the function, and
- In between parentheses, any input argument(s).

The function call can stand on its own, or can be on the right-hand-side of an assignment.

**Exercise 7.1.** Make the bisection algorithm more elegant by introducing functions `new_l`, `new_r` used as:

```
l = new_l(l, mid, fmid);
r = new_r(r, mid, fmid);
```

You can base this off the file `bisect.cpp` in the repository

Question: you could leave out `fmid` from the functions. Write this variant. Why is this not a good idea?

The function call

1. copies the value of the *function argument* to the *function parameter*;
2. causes the function body to be executed, and
3. the function call is replaced by whatever you `return`.
4. (If the function does not return anything, for instance because it only prints output, you declare the return type to be `void`.)

To introduce two formal concepts:

- A function definition can have zero or more *parameters*, or *formal parameters*. These function as variable definitions local to the function.
- The function call has a corresponding number of *arguments*, or *actual parameters*.

### 7.1.3 Why use functions?

In many cases, code that is written using functions can also be written without. So why would you use functions? There are several reasons for this.

Functions can be motivated as making your code more structured and intelligible. The source where you use the function call becomes shorter, and the function name makes the code more descriptive. This is sometimes called ‘self-documenting code’.

Sometimes introducing a function can be motivated from a point of *code reuse*: if the same block of code appears in two places in your source (this is known as *code duplication*), you replace this by one function definition, and two (single line) function calls.

Suppose you do the same computation twice:

```
double x, y, v, w;
y = ..... computation from x .....
w = ..... same computation, but from v .....
```

With a function this can be replaced by:

```
double computation(double in) {
    return .... computation from 'in' ....
}

y = computation(x);
w = computation(v);
```

Example: multiple norm calculations:

Repeated code:

```
float s = 0;
for (int i=0; i<x.size(); i++)
    s += abs(x[i]);
cout << "One norm x: " << s << endl;
s = 0;
for (int i=0; i<y.size(); i++)
    s += abs(y[i]);
cout << "One norm y: " << s << endl;
```

becomes:

```
float OneNorm( vector<float> a ) {
    float sum = 0;
    for (int i=0; i<a.size(); i++)
        sum += abs(a[i]);
    return sum;
}
int main() {
    ... // stuff
    cout << "One norm x: "
        << OneNorm(x) << endl;
    cout << "One norm y: "
        << OneNorm(y) << endl;
```

(Don’t worry about array stuff in this example)

A final argument for using functions is code maintainability:

- Easier to debug: if you use the same (or roughly the same) block of code twice, and you find an error, you need to fix it twice.
- Maintenance: if a block occurs twice, and you change something in such a block once, you have to remember to change the other occurrence(s) too.
- Localization: any variables that only serve the calculation in the function now have a limited scope.

```
void print_mod(int n,int d) {
    int m = n%d;
    cout << "The modulus of " << n << " and " << d
    << " is " << m << endl;
```

**Review 7.1.** True or false?

- The purpose of functions is to make your code shorter.
- Using functions makes your code easier to read and understand.
- Functions have to be defined before you can use them.
- Function definitions can go inside or outside the main program.

## 7.2 Anatomy of a function definition and call

Loosely, a function takes input and computes some result which is then returned. Just some simple examples:

```
int compute( float x, char c ) {
    /* code */
    return somevalue;
}
// in main:
i = compute(x,'c');
```

```
void compute( float x, char c ) {
    /* code */
}
// in main:
compute(x,'c');
```

So we need to discuss the function definition and its use.

## 7.3 Definition vs declaration

The C++ compiler translates your code in *one pass*: it goes from top to bottom through your code. This means you can not make reference to anything, such as a function name, that you haven't defined yet. For this reason, in the examples so far we put the function definition before the main program.

You don't have to put the full definition before the main program. for the compiler to judge whether a function call is legal it does not need the full function definition: it can proceed once it know the name of the function, and the types of the inputs and result. This information is sometimes called a *function header*, *function prototype*, or *function signature*, but the technical term is a *function declaration*.

In the following example we put the function declaration before the main program, and the full function definition after it:

Some people like the following style of defining a function:

```
// declaration before main
int my_computation(int);

int main() {
    int result;
    result = my_computation(5);
    return 0;
}

// definition after main
int my_computation(int i) {
    return i+3;
}
```

This is purely a matter of style.

See chapter 19 for more details.

## 7.4 Void functions

Some functions do not return a result value, for instance because only write output to screen or file. In that case you define the function to be of type `void`.

```
void print_header() {
    cout << "*****" << endl;
    cout << "* Output      *" << endl;
    cout << "*****" << endl;
}
int main() {
    print_header();
    cout << "The results for day 25:" << endl;
    // code that prints results ....
    return 0;
}
```

```
void print_result(int day, float value) {
    cout << "*****" << endl;
    cout << "* Output      *" << endl;
    cout << "*****" << endl;
    cout << "The results for day " << day << ":" << endl;
    cout << "      " << value << endl;
}
int main() {
    print_result(25, 3.456);
    return 0;
}
```

**Review 7.2.** True or false?

- A function can have only one input
- A function can have only one return result
- A void function can not have a `return` statement.

## 7.5 Parameter passing

C++ functions resemble mathematical functions: you have seen that a function can have an input and an output. In fact, they can have multiple inputs, separated by commas, but they have only one output.

$$a = f(x, y, i, j)$$

We start by studying functions that look like these mathematical functions. They involve a *parameter passing* mechanism called *passing by value*. Later we will then look at *passing by reference*.

### 7.5.1 Pass by value

The following style of programming is very much inspired by mathematical functions, and is known as *functional programming*<sup>1</sup>.

- A function has one result, which is returned through a return statement. The function call then looks like
- ```
y = f(x1, x2, x3);
```
- Example:

**Code :**

```
// func/passvalue.cpp
double squared( double x ) {
    double y = x*x;
    return y;
}
/* ... */
number = 5.1;
cout << "Input starts as: "
     << number << '\n';
other = squared(number);
cout << "Output var is: "
     << other << '\n';
cout << "Input var is now: "
     << number << '\n';
```

**Output :**

```
[func] passvalue:
Input starts as: 5.1
Output var is: 26.01
Input var is now: 5.1
```

- The definition of the C++ parameter passing mechanism says that input arguments are copied to the function, meaning that they don't change in the calling program:

---

1. There is more to functional programming. For instance, strictly speaking your whole program needs to be based on function calling. All code consists of function definitions and function calls.

```
Code:
// func/passvaluelocal.cpp
double squared( double x ) {
    x = x*x;
    return x;
}
/* ... */
number = 5.1;
cout << "Input starts as: "
    << number << '\n';
other = squared(number);
cout << "Output var is: "
    << other << '\n';
cout << "Input var is now: "
    << number << '\n';
```

```
Output:
[func] passvaluelocal:
Input starts as: 5.1
Output var is: 26.01
Input var is now: 5.1
```

We say that the input argument is *passed by value*: its value is copied into the function. In this example, the function parameter *x* acts as a local variable in the function, and it is initialized with a copy of the value of *number* in the main program.

**Exercise 7.2.** Write two functions

```
int biggest(int i,int j);
int smallest(int i,int j);
```

and a program that prints the results:

```
int i = 5, j = 17;
cout ... biggest(i,j) ...
cout ... smallest(i,j) ...
```

Passing a variable to a routine passes the value; in the routine, the variable is local. So, in this example the value of the argument is not changed:

```
Code:
// func/localparm.cpp
void change_scalar(int i) {
    i += 1;
}
/* ... */
number = 3;
cout << "Number is 3: "
    << number << '\n';
change_scalar(number);
cout << "is it still 3? Let's see: "
    << number << '\n';
```

```
Output:
[func] localparm:
Number is 3: 3
is it still 3? Let's see: 3
```

**Exercise 7.3.** If you are doing the prime numbers project (chapter 45) you can now do exercise 45.6.

**Exercise 7.4.** If you are doing the zero-finding project (chapter 47) you can now do exercise 47.9.

### 7.5.2 Pass by reference

Having only one output is a limitation on functions. You can also alter the input parameters and returning (possibly multiple) results that way. You do this by not copying values into the function parameters, but by turning the function parameters into aliases of the variables at the place where the function is called.

We need the concept of a *reference*: another variable to refers to the same ‘thing’ as another, already existing, variable.

A reference is indicated with an ampersand in its definition, and it acts as an alias of the thing it references.

**Code:**

```
// basic/ref.cpp
int i;
int &ri = i;
i = 5;
cout << i << "," << ri << '\n';
i *= 2;
cout << i << "," << ri << '\n';
ri -= 3;
cout << i << "," << ri << '\n';
```

**Output:**

```
[basic] ref:
5,5
10,10
7,7
```

(You will not use references often this way.)

Correct:

```
float x{1.5};
float &xref = x;
```

Not correct:

```
float x{1.5};
float &xref; // WRONG: needs to initialized immediately
xref = x;

float &threeref = 3; // WRONG: only reference to 'lvalue'
```

You can make a function parameter into a reference of a variable in the main program. This makes the function parameter into another name referring to the same thing.

The function parameter *n* becomes a reference to the variable *i* in the main program:

```
void f(int &n) {
    n = /* some expression */ ;
}
int main() {
    int i;
```

```
f(i);
// i now has the value that was set in the function
}
```

Using the ampersand, the parameter is *passed by reference*: instead of copying the value, the function receives a reference, so that the parameter becomes a reference to the thing in the *calling environment*.

**Remark** The pass by reference mechanism in C was *different and should not be used in C++*. In fact it was not a true pass by reference, but passing an address by value. If you do need the address of a variable, use `addressof` from the `memory` header, since the ampersand operator can be overloaded.

**Remark** We sometimes use the following terminology for function parameters:

- input parameters: passed by value, so that it only functions as input to the function, and no result is output through this parameter;
- output parameters: passed by reference so that they return an ‘output’ value to the program.
- throughput parameters: these are passed by reference, and they have an initial value when the function is called. C++, unlike Fortran, has no real separate syntax for these.

**Code:**

```
// basic/setbyref.cpp
void f( int &i ) {
    i = 5;
}
int main() {
    int var = 0;
    f(var);
    cout << var << '\n';
}
```

**Output:**

```
[basic] setbyref:
5
```

Compare the difference with leaving out the reference.

As an example, consider a function that tries to read a value from a file. With anything file-related, you always have to worry about the case of the file not existing and such. So our function returns:

- a boolean value to indicate whether the read succeeded, and
- the actual value if the read succeeded.

The following is a common idiom, where the success value is returned through the `return` statement, and the value through a parameter.

```
bool can_read_value( int &value ) {
    // this uses functions defined elsewhere
    int file_status = try_open_file();
    if (file_status==0)
        value = read_value_from_file();
    return file_status==0;
}

int main() {
    int n;
```

## 7. Functions

---

```
if (!can_read_value(n)) {
    // if you can't read the value, set a default
    n = 10;
}
..... do something with 'n' .....
```

We will learn better ways!

This latter example can also be solved, perhaps more idiomatically, with `std::optional`; section 24.6.2.

**Exercise 7.5.** Write a `void` function `swap` of two parameters that exchanges the input values:

**Code:**

```
// func/swap.cpp
int i=1, j=2;
cout << i << "," << j << '\n';
swap(i, j);
cout << i << "," << j << '\n';
```

**Output:**

```
[func] swap:
1,2
2,1
```

**Exercise 7.6.** Write a divisibility function that takes a number and a divisor, and gives:

- a `bool` return result indicating that the number is divisible, and
- a remainder as output parameter.

**Code:**

```
// func/divisible.cpp
cout << number;
if (is_divisible(number,
                 divisor, remainder))
    cout << " is divisible by ";
else
    cout << " has remainder "
        << remainder << " from ";
cout << divisor << '\n';
```

**Output:**

```
[func] divisible:
8 has remainder 2 from 3
8 is divisible by 4
```

**Exercise 7.7.** If you are doing the geometry project, you should now do the exercises in section 46.1.

## 7.6 Recursive functions

In mathematics, sequences are often recursively defined. For instance, the sequence of factorials  $n \mapsto f_n \equiv n!$  can be defined as

$$f_0 = 1, \quad \forall_{n>0}: f_n = n \times f_{n-1}.$$

Instead of using a subscript, we write an argument in parentheses

$$F(n) = \begin{cases} n \times F(n-1) & \text{if } n > 0 \\ 1 & \text{otherwise} \end{cases}$$

This is a form that can be translated into a C++ function. The header of a factorial function can look like:

```
int factorial(int n)
```

So what would the function body be? We need a `return` statement, and what we return should be  $n \times F(n - 1)$ :

```
int factorial(int n) {
    return n*factorial(n-1);
} // almost correct, but not quite
```

So what happens if you write

```
int f3; f3 = factorial(3);
```

Well,

- The expression `factorial(3)` calls the `factorial` function, substituting 3 for the argument  $n$ .
- The return statement returns `n*factorial(n-1)`, in this case `3*factorial(2)`.
- But what is `factorial(2)`? Evaluating that expression means that the `factorial` function is called again, but now with  $n$  equal to 2.
- Evaluating `factorial(2)` returns `2*factorial(1),...`
- ... which returns `1*factorial(0),...`
- ... which returns ...
- Uh oh. We forgot to include the case where  $n$  is zero. Let's fix that:

```
int factorial(int n) {
    if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

- Now `factorial(0)` is 1, so `factorial(1)` is `1*factorial(0)`, which is 1,...
- ... so `factorial(2)` is 2, and `factorial(3)` is 6.

**Exercise 7.8.** It is possible to define multiplication as repeated addition:

**Code:**

```
// func/mult.cpp
int times( int number, int mult ) {
    cout << "(" << mult << ")";
    if (mult==1)
        return number;
    else
        return number + times(number,mult-1);
}
```

**Output:**

[func] mult:

Enter number and multiplier  
recursive multiplication  
of 7 and 5: (5)(4)(3)(2)(1)35

Extend this idea to define powers as repeated multiplication.

You can base this off the file `mult.cpp` in the repository

**Exercise 7.9.** The *Egyptian multiplication* algorithm is almost 4000 years old. The result of multiplying  $x \times n$  is:

```

if n is even:
    twice the multiplication x × (n/2);
otherwise if n == 1:
    x
otherwise:
    x plus the multiplication x × (n - 1)

```

Extend the code of exercise 7.8 to implement this.

Food for thought: discuss the computational aspects of this algorithm to the traditional one of repeated addition.

**Exercise 7.10.** The sum of squares:

$$S_n = \sum_{n=1}^N n^2$$

can be defined recursively as

$$S_1 = 1, \quad S_n = n^2 + S_{n-1}.$$

Write a recursive function that implements this second definition. Test it on numbers that are input interactively.

Then write a program that prints the first 100 sums of squares.

How many squares do you need to sum before you get overflow? Can you estimate this number without running your program?

**Exercise 7.11.** Write a recursive function for computing Fibonacci numbers:

$$F_0 = 1, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

First write a program that computes  $F_n$  for a value  $n$  that is input interactively.

Then write a program that prints out a sequence of Fibonacci numbers; set interactively how many.

**Remark** If you let your Fibonacci program print out each time it computes a value, you'll see that most values are computed several times. (Math question: how many times?) This is wasteful in running time. This problem is addressed in section 66.3.1.

**Remark** A function does not need to call itself directly to be recursive; if it does so indirectly we can call this mutual recursion.

```

int f(int n) { return g(n-1); }
int g(int n) { return f(n); }

```

Now we have a problem:  $f$  refers to  $g$  before the latter is defined. See section 19.2 about forward declaration.

## 7.7 Other function topics

### 7.7.1 Default arguments

Functions can have *default argument(s)*:

```
double distance( double x, double y=0. ) {
    return sqrt( (x-y)*(x-y) );
}
...
d = distance(x); // distance to origin
d = distance(x,y); // distance between two points
```

Any default argument(s) should come last in the parameter list.

Don't trace a function unless I say so:

```
void dosomething(double x,bool trace=false) {
    if (trace) // report on stuff
};

int main() {
    dosomething(1); // this one I trust
    dosomething(2); // this one I trust
    dosomething(3,true); // this one I want to trace!
    dosomething(4); // this one I trust
    dosomething(5); // this one I trust
```

### 7.7.2 Polymorphic functions

You can have multiple functions with the same name:

```
double average(double a,double b) {
    return (a+b)/2;
}
double average(double a,double b,double c) {
    return (a+b+c)/3;
```

Distinguished by type or number of input arguments: can not differ only in return type.

```
int f(int x);
string f(int x); // DOES NOT WORK
```

### 7.7.3 Math functions

Some *math functions*, such as `abs`, can be included through the `cmath` header:

```
#include <cmath>
using std::abs;
```

The function `std::abs` is overloaded for different numeric types. Calling `abs` without that namespace prefix invokes an integer function `abs`, and the compiler may suggest that you use `fabs` for floating point arguments.

Others math functions, such as `max`, are in the less obvious `algorithm` header (see section 14.3):

```
#include <algorithm>
using std::max;
```

#### 7.7.4 Trailing return type

By analogy with the way lambda functions are defined:

```
[] ( /* args */ ) -> returntype { /* body */ } ;
```

it is also possible to use the *trailing return type* for regular function definitions:

```
// func/trail.cpp
auto times( int number, int mult ) -> int {
    if (mult==1)
        return number;
    else
        return number + times(number,mult-1);
}
```

#### 7.7.5 Stack overflow

So far you have seen only very simple recursive functions. Consider the function

$$\forall_{n>1}: g_n = (n-1) \cdot g(n-1), \quad g(1) = 1$$

and its implementation:

```
int multifact( int n ) {
    if (n==1)
        return 1;
    else {
        int oneless = n-1;
        return oneless*multifact(oneless);
    }
}
```

Now the function has a local variable. Suppose we compute  $g(3)$ . That involves

```
int oneless = 2;
```

and then the computation of  $g_2$ . But that computation involved

```
int oneless = 1;
```

Do we still get the right result for  $g_3$ ? Is it going to compute  $g_3 = 2 \cdot g_2$  or  $g_3 = 1 \cdot g_2$ ?

Not to worry: each time you call *multifact* a new local variable *oneless* gets created ‘on the stack’. That is good, because it means your program will be correct, but it also means that if your function has both

- a large amount of local data, and
- a large *recursion depth*,

it may lead to *stack overflow*.

**Remark** *Historical note: very old versions of Fortran did not do this, and so recursive functions were basically impossible.*

## 7.8 Review questions

**Review 7.3.** What is the output of the following programs? Assume that each program starts with

```
#include <iostream>
using std::cout;
using std::endl;

int add1(int i) {
    return i+1;
}
int main() {
    int i=5;
    i = add1(i);
    cout << i << endl;
}

void add1(int &i) {
    i = i+1;
}
int main() {
    int i=5;
    add1(i);
    cout << i << endl;
}
```

**Review 7.4.** Suppose a function

```
bool f(int);
```

is given, which is true for some positive input value. Write a main program that finds the smallest positive input value for which *f* is true.

**Review 7.5.** Suppose a function

```
bool f(int);
```

is given, which is true for some negative input value. Write a code fragment that finds the (negative) input with smallest absolute value for which *f* is true.

**Review 7.6.** Suppose a function

```
bool f(int);
```

is given, which computes some property of integers. Write a code fragment that tests if *f(i)* is true for some  $0 \leq i < 100$ , and if so, prints a message.

**Review 7.7.** Suppose a function

```
bool f(int);
```

is given, which computes some property of integers. Write a main program that tests if  $f(i)$  is true for all  $0 \leq i < 100$ , and if so, prints a message.

# Chapter 8

## Scope

### 8.1 Scope rules

The concept of *scope* answers the question ‘when is the binding between a name (read: variable) and the internal entity valid’.

#### 8.1.1 Lexical scope

C++, like Fortran and most other modern languages, uses *lexical scope* rule. This means that you can textually determine what a variable name refers to.

```
int main() {
    int i;
    if ( something ) {
        int j;
        // code with i and j
    }
    int k;
    // code with i and k
}
```

- The lexical scope of the variables *i, k* is the main program including any blocks in it, such as the conditional, from the point of definition onward. You can think that the variable in memory is created when the program execution reaches that statement, and after that it can be referred to by that name.
- The lexical scope of *j* is limited to the true branch of the conditional. The integer quantity is only created if the true branch is executed, and you can refer to it during that execution. After execution leaves the conditional, the name ceases to exist, and so does the integer in memory.
- In general you can say that any *use* of a name has to be in the lexical scope of that variable, and after its *definition*.

#### 8.1.2 Shadowing

Scope can be limited by an occurrence of a variable by the same name:

## 8. Scope

---

**Code:**

```
// basic/shadowtrue.cpp
bool something{true};
int i = 3;
if ( something ) {
    int i = 5;
    cout << "Local: " << i << '\n';
}
cout << "Global: " << i << '\n';
if ( something ) {
    float i = 1.2;
    cout << "Local again: " << i << '\n';
}
cout << "Global again: " << i << '\n';
```

**Output:**

```
[basic] shadowtrue:
Local: 5
Global: 3
Local again: 1.2
Global again: 3
```

The first variable `i` has lexical scope of the whole program, minus the two conditionals. While its *lifetime* is the whole program, it is unreachable in places because it is *shadowed* by the variable `i` in the conditionals.

This is independent of dynamic / runtime behavior!

**Exercise 8.1.** What is the output of this code?

```
// basic/shadowfalse.cpp
bool something{false};
int i = 3;
if ( something ) {
    int i = 5;
    cout << "Local: " << i << '\n';
}
cout << "Global: " << i << '\n';
if ( something ) {
    float i = 1.2;
    cout << i << '\n';
    cout << "Local again: " << i << '\n';
}
cout << "Global again: " << i << '\n';
```

**Exercise 8.2.** What is the output of this code?

```
for ( int i=0; i<2; i++ ) {
    int j; cout << j << endl;
    j = 2; cout << j << endl;
}
```

### 8.1.3 Lifetime versus reachability

The use of functions introduces a complication in the lexical scope story: a variable can be present in memory, but may not be textually accessible:

```
void f() {
    ...
```

```
}
```

```
int main() {
    int i;
    f();
    cout << i;
}
```

During the execution of `f`, the variable `i` is present in memory, and it is unaltered after the execution of `f`, but it is not accessible.

A special case of this is recursion:

```
void f(int i) {
    int j = i;
    if (i<100)
        f(i+1);
}
```

Now each incarnation of `f` has a local variable `i`; during a recursive call the outer `i` is still alive, but it is inaccessible.

## 8.1.4 Scope subtleties

### 8.1.4.1 Forward declaration

If you have two functions `f`, `g` that call each other,

```
int f(int i) { return g(i-1); }
int g(int i) { return f(i+1); }
```

you need a *forward declaration*

```
int g(int);
int f(int i) { return g(i-1); }
int g(int i) { return f(i+1); }
```

since the use of name `g` has to come after its declaration.

There is also forward declaration of *classes*. You can use this if one class contains a pointer to the other:

```
class B;
class A {
private:
    shared_ptr<B> myB;
};
class B {
private:
    int myint;
}
```

You can also use a forward declaration if one class is an argument or return type:

```
class B;
class A {
public:
    B GimmeA();
```

## 8. Scope

---

```
};

class B {
public:
    B(int);
}
```

However, there is a subtlety here: in the definition of `A` you can not give the full definition of the function that return `B`:

```
class B;
class A {
public:
    B GimmeaB() { return B(5); }; // WRONG: does not compile
};
```

because the compiler does not yet know the form of the `B` constructor.

The right way:

```
class B;
class A {
public:
    B GimmeaB();
};

class B {
public:
    B(int);
}

A::GimmeaB() { return B(5); };
```

### 8.1.4.2 Closures

The use of lambdas or *closures* (chapter 13) comes with another exception to general scope rules. Read about ‘capture’.

## 8.2 Static variables

Variables in a function have *lexical scope* limited to that function. Normally they also have *dynamic scope* limited to the function execution: after the function finishes they completely disappear. (Class objects have their *destructor* called.)

There is an exception: a *static variable* persists between function invocations.

```
void fun() {
    static int remember;
}
```

For example

```
int onemore() {
    static int remember++; return remember;
}
int main() {
```

```
for ( ... )
    cout << onemode() << end;
return 0;
}
```

gives a stream of integers.

**Exercise 8.3.** The static variable in the `onemode` function is never initialized. Can you find a mechanism for doing so? Can you do it with a default argument to the function?

### 8.3 Scope and memory

The notion of scope is connected to the fact that variables correspond to objects in memory. Memory is only reserved for an entity during the dynamic scope of the entity. This story is clear in simple cases:

```
int main() {
    // memory reserved for 'i'
    if (true) {
        int i; // now reserving memory for integer i
        ... code ...
    }
    // memory for 'i' is released
}
```

Recursive functions offer a complication:

```
int f(int i) {
    int itmp;
    ... code with 'itmp' ...
    if (something)
        return f(i-1);
    else return 1;
}
```

Now each recursive call of `f` reserves space for its own incarnation of `itmp`.

In both of the above cases the variables are said to be on the *stack*: each next level of scope nesting or recursive function invocation creates new memory space, and that space is released before the enclosing level is released.

Objects behave like variables as described above: their memory is released when they go out of scope. However, in addition, a *destructor* is called on the object, and on all its contained objects:

## 8. Scope

---

**Code:**

```
// object/destructor.cpp
class SomeObject {
public:
    SomeObject() {
        cout << "calling the constructor"
        << '\n';
    };
    ~SomeObject() {
        cout << "calling the destructor"
        << '\n';
    };
};
```

**Output:**

[object] **constructor**:

Before the nested scope

calling the constructor

Inside the nested scope

calling the destructor

After the nested scope

### 8.4 Review questions

**Review 8.1.** Is this a valid program?

```
void f() { i = 1; }
int main() {
    int i=2;
    f();
    return 0;
}
```

If yes, what does it do; if no, why not?

**Review 8.2.** What is the output of:

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
    int i=5;
    if (true) { i = 6; }
    cout << i << endl;
    return 0;
}
```

**Review 8.3.** What is the output of:

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
    int i=5;
    if (true) { int i = 6; }
    cout << i << endl;
    return 0;
}
```

{}

**Review 8.4.** What is the output of:

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
    int i=2;
    i += /* 5;
i += */ 6;
    cout << i << endl;
    return 0;
}
```

## 8. Scope

---

# Chapter 9

## Classes and objects

### 9.1 What is an object?

You have now learned about elementary data, control structures, and functions. Ultimately, that's all there is to programming: data and operations on them. However, to keep your programs manageable it is a good idea to structure them, and recognize that you really want to talk at a higher level of abstraction.

C++ offers an important mechanism of unifying data and operations to give a new level of abstraction: *objects* belonging to *classes*.

An object is an entity that you can request to do certain things.

When designing a class, first ask yourself: 'what functionality should the objects support'.

- The actions an object is capable of are the *methods* or *function members* of the object; and
- to make these actions possible the object probably stores data, the *data members*.
- Objects come in *classes*. A class is like a datatype: you can make objects of a class like you make variables of a datatype.
- Objects of the same class have the same methods. They also have the same members, but with individual values.

Classes are like datatypes in that you can declare variables of that type, which can then be used in expressions. Unlike basic datatypes, they are not predefined, so you first need to define the class before you can make objects of that class.

- You need a class definition, typically placed before the main program.
- (In larger programs you would put it in a *include file* or *module*.)
- You can then declare multiple objects belonging to that class.
- Objects can then be used in expressions, passed as parameter, et cetera.

#### 9.1.1 First example: points in the plane

In this section we will use a simple example as an illustration of how to use object: we are going to create a *Point* object, corresponding to a mathematical point in  $\mathbb{R}^2$ .

The presentation will be rather top-down: we will more focus on what to do, rather than how to do it. No worries, all details will be covered in due time.

## 9. Classes and objects

---

**Exercise 9.1.** Thought exercise: what are some of the actions that a point object should be capable of?

The first things we are going to do with a point are to query some of its mathematical properties. For instance, given a point, you could want to know its distance to the origin or its angle with the  $x$ -axis.

Small illustration: point objects.

**Code:**

```
// object/functionality.cpp
Point p(1.,2.);
cout << "distance to origin "
    << p.distance_to_origin()
    << '\n';
p.scaleby(2.);
cout << "distance to origin "
    << p.distance_to_origin()
    << '\n'
    << "and angle " << p.angle()
    << '\n';
```

**Output:**

[object] functionality:

```
distance to origin 2.23607
distance to origin 4.47214
and angle 1.10715
```

Note the ‘dot’ notation.

**Exercise 9.2.** Thought exercise:

What data does the object need to store to be able to calculate angle and distance to the origin?  
Is there more than one possibility?

Food for thought: you may be tempted to write methods for getting the  $x$  and  $y$  coordinate. However, ask yourself if those should be publicly visible methods. Is getting the  $x$  coordinate a linear algebra operation? When you have methods such as ‘get the distance to the origin’ or ‘shift this point rightward’, do you explicitly need the coordinates?

The above example used a `Point` object without saying how it was created. That’s what we are going to look at next.

- First define the class, with data and function members:

```
class MyObject {
    // define class members
    // define class methods
};
```

(details later) typically before the `main`.

- You create specific objects with a declaration

```
MyObject
    object1( /*..*/ ),
    object2( /*..*/ );
```

- You let the objects do things:

```
object1.do_this();
```

```
x = object2.do_that( /* ... */ );
```

Let's now introduce the details of all these steps.

### 9.1.2 Constructor

First we'll look at creating class objects, and we'll stick with the point example.

Since a point can be defined by its  $x, y$  coordinates, you can imagine that

- the point object stores these coordinates, and
- when you create a point object, you do that by specifying the coordinates.

Here are the relevant fragments of code:

The declaration of an object `x` of class `Point`; the coordinates of the point are initially set to `1.5, 2.5`.

```
Point x(1.5, 2.5);
```

```
class Point {
    private: // data members
        double x, y;
    public: // function members
    Point
        ( double x_in, double y_in ) {
            x = x_in; y = y_in;
        };
        /* ... */
};
```

Study the implementation closely. The class is named `Point`, and you see something that looks like a function definition, also named `Point`. However, unlike a regular function, it does not have a return type, not even `void`.

This function is named the *constructor* of the class, and it is characterized by:

- The constructor has the same name as the class, and
- it looks like a function definition, except that it has no return type.

When you create an object, in the manner you've seen in above examples, you actually call this constructor.

Usually you write your own constructor, for instance to initialize data members. In the case of the `class Point` the function of the constructor is to take the coordinates of the point and to copy them to private members of the `Point` object.

If the object you create can have sensible default values, you can also use a *default constructor*, which has no argument. We will get to that below; section 9.1.7.

### 9.1.3 Data members

In the examples so far, you created a point object from its coordinates

```
Point oneone(1., 1.);
```

## 9. Classes and objects

---

and the `Point` object stored these coordinates. However, this connection between outward usage and internal implementation can be very different. Maybe you have an application that works in polar coordinates, in which case storing  $r, \theta$  is more natural, or at least more convenient for computation. But you may still want to create a point from its Cartesian coordinates.

The arguments of the constructor imply nothing about what data members are stored!

Example: create a point in where the constructor uses  $x, y$  Cartesian coordinates, but which internally stores  $r, \theta$  polar coordinates:

```
#include <cmath>
class Point {
private: // members
    double r, theta;
public: // methods
    Point( double x, double y ) {
        r = sqrt(x*x+y*y);
        theta = atan2(y, x);
    }
}
```

Note: no change to outward API.

You have now seen the `private` and `public` keywords. These indicate the visibility of class members.

- Keyword `private` indicates that data is internal: not accessible from outside the object; can only be used inside function members.
- Keyword `public` indicates that the constructor function can be used in the program.

You may have observed that the private members are all data members, while all the function members are public. This is no coincidence: for now you can consider this a ‘best practice’.

Best practice we will use:

```
class MyClass {
private:
    // data members
public:
    // methods
}
```

- Data is private: not visible outside of the objects.
- Methods are public: can be used in the code that uses objects.
- You can have multiple private/public sections, in any order.

### 9.1.4 Methods

Methods are things you can ask your class objects to do. For instance, in the `Point` class, you could ask a point to report its distance to the origin, or you could ask it to scale its distance by some number.

Let’s start with the simpler of these two: measuring the distance to the origin. Without classes and objects, you would write a function with  $x, y$  coordinates as input, and a single number as output

```
float x = ..., y = ...;
float d = distance_to_origin(x, y);
```

For an object method this looks like:

```
float x=..., y=...;  
Point p( x,y );  
float d = p.distance_to_origin();
```

To point out differences and similarities:

- You're still using a function with a scalar output, but
- instead of input parameters we use the coordinates that are stored in the point object. These act as 'global variables', at least within the object.
- To apply this function to a point, we use the 'dot' notation. You could pronounce this as 'p's distance to the origin'. Note that this function is only available as applying to a point; you can not use it in other contexts.

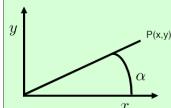
Definition and use of the *distance* function:

**Code:**

```
// geom/pointclass.cpp
class Point {
private:
    float x,y;
public:
    Point(float in_x,float in_y) {
        x = in_x; y = in_y; }
    float distance_to_origin() {
        return sqrt( x*x + y*y );
    }
}
/* ... */
Point p1(1.0,1.0);
float d = p1.distance_to_origin();
cout << "Distance to origin: "
     << d << '\n';
```

**Output:**  
[geom] **pointclass**:  
Distance to origin: 1.41421

**Exercise 9.3.** Add a method *angle* to the *Point* class. How many parameters does it need?



Hint: use the function *atan* or *atan2*.

You can base this off the file *pointclass.cpp* in the repository

**Exercise 9.4.** Make a class *GridPoint* for points that have only integer coordinates. Implement a function *manhattan\_distance* which gives the distance to the origin counting how many steps horizontal plus vertical it takes to reach that point.

## 9.1.5 Initialization

There are various ways of setting the initial values of the data members of an object.

## 9. Classes and objects

---

### 9.1.5.1 Default values

Sometimes it makes sense for objects to have default values if nothing else is specified. This can be done with setting default values on the data members:

Class members can have default values, just like ordinary variables:

```
class Point {  
private:  
    float x=3., y=.14;  
public:  
    // et cetera  
}
```

Each object will have its members initialized to these values.

### 9.1.5.2 Initialization in the constructor

Above you saw some examples of constructors that are used to initialize the object data. Initialization can happen in a number of ways.

First of all, you can copy the constructor arguments in the body of the constructor. However, the preferred way is by using *member initializers*, which takes a new notation.

The naive way:

```
class Point {  
private:  
    float x,y;  
public:  
    Point( float in_x,  
           float in_y ) {  
        x = in_x; y = in_y;  
    };
```

The preferred way:

```
// geom/pointinit.cpp  
class Point {  
private:  
    float x,y;  
public:  
    Point( float in_x,  
           float in_y )  
        : x(in_x),y(in_y) {  
    }
```

Explanation later. It's technical.

(See section 10.6 for why member initializers are preferred.)

You can even save yourself from having to think of too many names:

```
Code:
// geom/pointinitxy.cpp
class Point {
private:
    float x,y;
public:
    Point( float x, float y )
        : x(x),y(y) {
    }
    /* ... */
    Point p1(1.,2.);
    cout << "p1 = "
        << p1.getx() << "," << p1.gety()
        << '\n';
}
```

```
Output:
[geom] pointinitxy:
p1 = 1,2
```

The initialization `x(x)` should be read as `membername(argumentname)`. Yes, having `x` twice is a little confusing.

### 9.1.6 Methods, a deeper dive

You have just seen examples of class *methods*: a function that is only defined for objects of that class, and that has access to the private data of that object.

In exercise 9.3 you implemented an `angle` function, that computed the angle from the stored coordinates. You could have made other decisions.

**Exercise 9.5.** Discuss the pros and cons of this design:

```
class Point {
private:
    double x,y,r,theta;
public:
    Point(double xx,double yy) {
        x = xx; y = yy;
        r = // sqrt something
        theta = // something trig
    };
    double angle() { return theta; };
};
```

By making these functions public, and the data members private, you define an Application Programmer Interface (API) for the class:

- You are defining operations for that class; they are the only way to access the data of the object.
- The methods can use the data of the object, or alter it. All data members, even when declared `private`, are global to the methods.
- Data members declared `private` are not accessible from outside the object.

**Review 9.1.** T/F?

- A class is primarily determined by the data it stores. Class determined by its data+
- A class is primarily determined by its methods. Class determined by its methods+
- If you change the design of the class data, you need to change the constructor call. Change data, change constructor proto too+

Now let's look at some different types of objects. This is an informal classification, not necessarily corresponding to defined concepts in the C++ standard.

#### 9.1.6.1 *Changing state*

Objects usually have data members that maintain the *state* of the object. By changing the values of the members you change the state of the object. Doing so is usually done through a method.

For instance, you may want to scale a vector by some amount:

**Code:**

```
// geom/pointscaleby.cpp
class Point {
    /* ... */
    void scaleby( float a ) {
        x *= a; y *= a; }
    /* ... */
};

/* ... */
Point p1(1.,2.);
cout << "p1 to origin "
    << p1.distance_to_origin()
    << '\n';
p1.scaleby(2.);
cout << "p1 to origin "
    << p1.distance_to_origin()
    << '\n';
```

**Output:**

```
[geom] pointscaleby:
p1 to origin 2.23607
p1 to origin 4.47214
```

**Exercise 9.6.** Implement a method *shift\_right* for the *Point* class.

**Exercise 9.7.** Take the *Point* class design that uses polar coordinates (see above). Implement a *rotate* method.

There is a subtlety here. Hint: imagine rotating a point sufficiently many times.

#### 9.1.6.2 *Methods that return objects*

The methods you have seen so far only returned elementary datatypes. It is also possible to return an object, even from the same class. For instance, instead of scaling the members of a vector object, you could create a new object based on the scaled members:

**Code:**

```
// geom/pointscale.cpp
class Point {
    /* ... */
    Point scale( float a ) {
        Point scaledpoint( x*a, y*a );
        return scaledpoint;
    };
    /* ... */
    println("p1 to origin {:.5}",
           p1.dist_to_origin());
    Point p2 = p1.scale(2.);
    println("p2 to origin {:.5}",
           p2.dist_to_origin());
```

**Output:**

```
[geom] pointscale:
p1 to origin 2.2361
p2 to origin 4.4721
```

Create a point by scaling another point:

```
new_point = old_point.scale(2.81);
```

Two ways of handling the `return` statement of the `scale` method:

Naive:

```
// geom/pointscale.cpp
Point Point::scale( float a ) {
    Point scaledpoint =
        Point( x*a, y*a );
    return scaledpoint;
};
```

Creates point, copies it to `new_point`

Better:

```
// geom/pointscale.cpp
Point Point::scale( float a ) {
    return Point( x*a, y*a );
};
```

Creates point, moves it directly to `new_point`

'move semantics' and 'copy elision': compiler is pretty good at avoiding copies

### 9.1.7 Default constructor

You have now seen some examples of classes and their constructors. These constructors took arguments that set the initial state of the object.

However, if your objects have sensible default values, you can use a *default constructor*. For example:

No constructor explicitly defined;

You recognize the default constructor in the main by the fact that an object is defined without any parameters.

## 9. Classes and objects

---

**Code:**

```
// object/default.cpp
class IamOne {
private:
    int i=1;
public:
    void print() {
        cout << i << '\n';
    }
};

/* ...
IamOne one;
one.print();
```

**Output:**

```
[object] defaultno:
1
```

You can define a default constructor yourself, but the previous example had a *defaulted* default constructor: it acted like it had a constructor

```
IamZero() {};
```

Bear this in mind as you study the following code:

```
// geom/pointdefault.cpp
Point p1(1.,2.), p2;
cout << "p1 to origin " << p1.length() << '\n';
p2 = p1.scale(2.);
cout << "p2 to origin " << p2.length() << '\n';
```

With the *Point* class (and its constructor) as given before:

```
// geom/pointclass.cpp
class Point {
private:
    float x,y;
public:
    Point(float in_x,float in_y) {
        x = in_x; y = in_y; }
    float distance_to_origin() {
        return sqrt( x*x + y*y );
    }
};

/* ...
Point p1(1.0,1.0);
float d = p1.distance_to_origin();
cout << "Distance to origin: "
<< d << '\n';
```

this will give an error message during compilation. The reason is that

```
Point p2;
```

calls the default constructor. Now that you have defined your own constructor, the default constructor no longer exists. So you need to define it explicitly:

```
// geom/pointdefault.cpp
Point() = default;
Point( float x,float y )
    : x(x),y(y) {};
```

You now have a class with two constructors. The compiler will figure out which one to use. This is an example of *polymorphism*.

You can also indicate somewhat more explicitly that the *defaulted default constructor* needs to exist:

```
// object/default.cpp
Point() = default;
Point( double x, double y )
: x(x), y(y) {};
```

**Remark** The default constructor has ‘empty parentheses’, but you use it specifying no parentheses. What would happen if you specified empty parentheses when you create an object?

```
// object/constructparen.cpp
class MyClass {
public:
    MyClass() { cout << "Construct!" << '\n'; };
};

int main() {

    MyClass x;
    MyClass y();

    constructparen.cpp:24:12: warning:
        empty parentheses interpreted as a function declaration
            MyClass y();
                           ^
    constructparen.cpp:24:12: note:
        remove parentheses to declare a variable
            MyClass y();
                           ^
1 warning generated.
```

### 9.1.8 Data member access; invariants

You may have noticed the keywords `public` and `private`. We made the data members private, and the methods public. The C++ language also has the `struct` construct, inherited from C. In that one, data members are (by default) public. Why don’t we do that here?

Struct data is public:

```
struct Point {
    double x;
};
int main() {
    Point andhalf;
    andhalf.x = 1.5;
}
```

```
class Point {
public: // Bad! Idea!
    double x;
};
int main() {
    Point andhalf;
    andhalf.x = 2.6;
}
```

Objects are really supposed to be accessed through their functionality. While you could write methods such as `get_x`, (this is called an *accessor*; see also section 18.4 for some subtleties) to get the *x* coordi-

nate, ask yourself if that makes sense. If you need the  $x$  coordinate to shift the point rightward, write a `shift_right` method instead.

- Interface: `public` functions that determine the functionality of the object; effect on data members is secondary.
- Implementation: data members, keep `private`: they only support the functionality.

This separation is a Good Thing:

- Protect yourself against inadvertent changes of object data.
- Possible to change implementation without rewriting calling code.

You should not write access functions lightly: you should first think about what elements of your class should conceptually be inspectable or changeable by the outside world. Consider for example a class where a certain relation holds between members. In that case only changes are allowed that maintain that relation. It is sometimes said that a class satisfies an *invariant*.

You already saw this phenomenon in action in exercise 9.7. What was the invariant there? Let's consider another example of the need of maintaining an invariant.

We make a class for points on the unit circle

```
// object/unit.cpp
class UnitCirclePoint {
private:
    float x, y;
public:
    UnitCirclePoint(float x) {
        setx(x); }
    void setx(float newx) {
        x = newx; y = sqrt(1-x*x);
    };
}
```

You don't want to be able to change just one of  $x, y$ !

In general: enforce invariants on the members.

Section 9.5.5 has some further discussion on ways of directly accessing internal data.

### 9.1.9 Examples

So far, we have looked at examples of objects that represent ‘object-like’ things in the real world. However, we can also make objects for things that are more abstract. In the next example, we look at ‘infinite objects’, such as the set of all integers. Clearly, it is not possible to store an infinite amount of data, but we don’t need to. Instead we ask: what are the methods for an object that is the set of all integers? One possible design is that you could ask this object ‘give me the next integer’.

Objects can model fairly abstract things:

|                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                       |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| <b>Code:</b><br><pre>// object/stream.cpp class Stream { private:     int last_result{0}; public:     int next() {         return last_result++;     }      int main() {         Stream ints;         cout &lt;&lt; "Next: "             &lt;&lt; ints.next() &lt;&lt; '\n';         cout &lt;&lt; "Next: "             &lt;&lt; ints.next() &lt;&lt; '\n';         cout &lt;&lt; "Next: "             &lt;&lt; ints.next() &lt;&lt; '\n';     } }</pre> | <b>Output:</b><br><pre>[object] stream: Next: 0 Next: 1 Next: 2</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|

**Exercise 9.8.**

- Write a class `multiples_of_two` where every call of `next` yields the next multiple of two.
- Write a class `multiples` used as follows:

`multiples multiples_of_three(3);`

where the `next` call gives the next multiple of the argument of the constructor.

You can base this off the file `stream.cpp` in the repository

**Exercise 9.9.** If you are doing the prime project (chapter 45), now is a good time to do exercise in section 45.6.**9.2 Inclusion relations between classes**

The data members of an object can be of elementary datatypes, or they can be objects. For instance, if you write software to manage courses, each `Course` object will likely have a `Person` object, corresponding to the teacher.

```
class Person {
    string name;
    ...
}

class Course {
private:
    int year;
    Person the_instructor;
    vector<Person> students;
}
```

Designing objects with relations between them is a great mechanism for writing structured code, as it makes the objects in code behave like objects in the real world. The relation where one object contains another, is called a *has-a* relation between classes.

### 9.2.1 Literal and figurative has-a

Sometimes a class can behave as if it includes an object of another class, while not storing this other object. Consider the example of a line segment, that is, the segment from a starting point to an ending point. We want to offer the API:

```
int main() {
    Segment somesegment( /* something */ );
    Point somepoint = somesegment.get_the_end_point();
```

We can support this by letting the *Segment* class actually store the starting and ending points:

```
class Segment {
private:
    Point starting_point, ending_point;
}
```

or letting it store a distance and angle from the starting point:

```
class Segment {
private:
    Point starting_point;
    float length, angle;
}
```

In both cases the code using the object is written as if the segment object contains two points. This illustrates how object-oriented programming can decouple the API of classes from their actual implementation.

Related to this decoupling, a class can also have two very different constructors.

```
class Segment {
private:
    // up to you how to implement!
public:
    Segment( Point start, float length, float angle )
    { ... }
    Segment( Point start, Point end ) { ... }
```

Depending on how you actually implement the class, the one constructor will simply store the defining data, and the other will do some conversion from the given data to the actually stored data.

This is another strength of object-oriented programming: you can change your mind about the implementation of a class without having to change the program that uses the class.

When you have a has-a relation between classes, the ‘default constructor’ problem (section 9.1.7) can pop up again:

Class for a person:

```
class Person {
private:
    string name;
public:
    Person( string name ) {
        /* ... */
    };
};
```

Class for a course, which contains a person:

```
class Course {
private:
    Person instructor;
    int enrollment;
public:
    Course( string instr, int n ) {
        /* ???? */
    };
};
```

Declare a `Course` variable as: `Course("Eijkhout", 65);`

Possible constructor:

```
Course( string teachername, int nstudents ) {
    instructor = Person(teachername);
    enrollment = nstudents;
};
```

Preferred:

```
Course( string teachername, int nstudents )
    : instructor(Person(teachername)),
    enrollment(nstudents) {
};
```

```
class Inner { /* ... */ };
class Outer {
private:
    Inner inside_thing;
```

Two possibilities for constructor:

```
Outer( Inner thing )
    : inside_thing(thing) {};
```

The `Inner` object is copied during construction of `Outer` object.

```
Outer( Inner thing )
    inside_thing = thing;
};
```

The `Outer` object is created, including construction of `Inner` object, then the argument is copied into place:  $\Rightarrow$  needs default constructor on `Inner`.

**Exercise 9.10.** If you are doing the geometry project, this is a good time to do the exercises in section 46.3.

### 9.2.1.1 Shorthand for objects

For classes with a constructor, you can use a shorthand for an object, giving a brace-delimited initializer list.

An *initializer list* can be used as a denotation.

```
// geom/rectcurly.cpp
Point(float ux, float uy) {
    /* ... */
}
Rectangle(Point bl, Point tr) {
    /* ... */
}
Point origin{0.,0.};
Rectangle lielow( origin, {5,2} );
```

## 9.3 Inheritance

In addition to the has-a relation, there is the *is-a relation*, also called *inheritance*. Here one class is a special case of another class. Typically the object of the *derived class* (the special case) then also inherits the data and methods of the *base class* (the general case).

General *FunctionInterpolator* class with method `value_at`. Derived classes:

- *LagrangeInterpolator* with `add_point_and_value`;
- *HermiteInterpolator* with `add_point_and_derivative`;
- *SplineInterpolator* with `set_degree`.

How do you define a derived class? The general code schema for use of a base class and derived class goes like this:

Base class, general case:

```
class General {
protected: // note!
    int g;
public:
    void general_method() {};
};
```

Derived class, special case:

```
class Special : public General {
public:
    void special_method() {
        ... g ...
    };
};
```

These are the various aspects of declaring a derived class:

- You need to indicate what the base class is:

```
class Special : public General { .... }
```

- The base class needs to declare its data members as `protected`: this is similar to private, except that they are visible to derived classes
- The methods of the derived class can then refer to data of the base class;
- Any method or data member defined for the base class is available for a derived class object.

The derived class has its own constructor, with the same name as the class name, but when it is invoked, it also calls the constructor of the base class. This can be the default constructor, but often you want to call the base constructor explicitly, with parameters that are describing how the special case relates to the base case.

In the following example, we have a general case, depending on two independent parameters. The special case comes from having a certain relationship between these parameters.

```
class General {
public:
    General( double x, double y ) {};
};

class Special : public General {
public:
    Special( double x ) : General(x, x+1) {};
};
```

Methods and data can be

- `private`, because they are only used internally;
- `public`, because they should be usable from outside a class object, for instance in the main program;
- `protected`, because they should be usable in derived classes.

**Exercise 9.11.** If you are doing the geometry project, you can now do the exercises in section 46.4.

### 9.3.1 Methods of base and derived classes

Above, it was assumed that derived classes use the methods of the base class unchanged. Sometimes, however, you may want the derived class have a different version of a method. This is done through the `virtual` and `override` keywords.

- A derived class can inherit a method from the base class.
- A derived class can define a method that the base class does not have.
- A derived class can *override* a base class method:

```
class Base {
public:
    virtual f() { ... };
};

class Deriv : public Base {
public:
    virtual f() override { ... };
};
```

### 9.3.2 Virtual methods

The methods of base and derived classes can relate in a number of ways.

- Method defined in base class: can be used in any derived class.
  - Method define in derived class: can only be used in that particular derived class.
  - Method defined both in base and derived class, marked `override`: derived class method replaces (or extends) base class method.
  - Virtual method: base class only declares that a routine has to exist, but does not give base implementation.  
A class is called *abstract class* if it has virtual methods; pure virtual if all methods are virtual.
- You can not make abstract objects.

Syntax for pure virtual method:

```
// object/purevirtual.cpp
class Base {
protected:
    int i;
public:
    Base(int i) : i(i) {};
    virtual ~Base() {};
    virtual int thevalue() = 0;
};

class Deriv : public Base {
public:
    Deriv(int i) : Base(i) {};
    virtual int thevalue() override {
        return 25;
    };
};
```

Because of the virtual function you can not create objects of the base class, only of the derived class.

The next example is of a *pure virtual* class: all methods are virtual. (Other programming languages may call this an *interface*.) This is a way of enforcing an API on derived classes.

Use case:

```
// object/virtualvec.hpp
class VirtualVector {
private:
public:
    virtual void setlinear(float) = 0;
    virtual float operator[](int) = 0;
};
```

Suppose `DenseVector` derives from `VirtualVector`:

```
// object/densevec.cpp
DenseVector v(5);
v.setlinear(7.2);
printf("%f", v[3]);
```

In the above codes, classes that have virtual methods have a virtual destructor. Otherwise, the derived class' destructor will not be called if the object is deleted through a pointer to the base class.

For completeness we give the implementation of the derived class of this pure virtual class.

Implementation:

Class:

```
// object/densevec.cpp
class DenseVector : VirtualVector {
private:
    vector<float> values;
public:
    DenseVector( int size ) {
        values = vector<float>(size,0);
    };
}
```

Methods:

```
// object/densevec.cpp
void setlinear( float v ) {
    for (int i=0; i<values.size(); ++i)
        values[i] = i*v;
}
float operator[](int i) {
    return values[i];
}
};
```

**Exercise 9.12.** Write a small ‘integrator’ library for Ordinary Differential Equations (ODEs)s. Assuming ‘autonomous ODEs’, that is  $u' = f(t)$  with no  $u$ -dependence, there are two simple integration schemes:

- explicit:  $u_{n+1} = u_n + \Delta t f_n$ ; and
- implicit:  $u_{n+1} = u_n + \Delta t f_{n+1}$ .

Write an abstract *Integrator* class where the *nextstep* method (which integrates for another  $\Delta t$ ) is pure virtual; then write *ExplicitIntegrator* and *ImplicitIntegrator* classes deriving from this.

```
// ode/pureint.cpp
double stepsize = .01;
auto integrate_linear =
    ForwardIntegrator( [] (double x) { return x*x; }, stepsize );
double int1 = integrate_linear.to(1.);
```

You can hardcode the function to be integrated, or try to pass a lambda expression or functor.

### 9.3.3 Friend classes

A *friend* class can access private data and methods even if there is no inheritance relationship.

```
/* forward definition: */
class A;
class B {
    // A objects can access B internals:
    friend class A;
private:
    int i;
};
class A {
public:
    void f(B b) { b.i; }; // friend access
};
```

### 9.3.4 Multiple inheritance

- Multiple inheritance: an X is-a A, but also is-a B.  
This mechanism is somewhat dangerous.
- Virtual base class: you don’t actually define a function in the base class, you only say ‘any derived class has to define this function’.

**Exercise 9.13.** If you are doing the geometry project, this is a good time to do the exercises in section 46.2.

## 9.4 More about constructors

### 9.4.1 Delegating constructors

If you have two constructors where one is a special case of the other, you can elegantly express that with *delegating constructors*.

As an example, consider a class that contains a vector, and you want to have the choice whether to create that as an empty vector, or to initialize it from another vector:

Two constructors:

```
// object/delegate.cpp
class MyVector {
private:
    vector<float> data;
public:
    MyVector( int nsize );
    MyVector( vector<float> indata );
};
```

You could of course write two separate constructors (here we use a member-initializer list in only one of them):

```
// object/delegate.cpp
// simple-minded way
MyVector::MyVector( int nsize )
    : data( vector<float>(nsize) ) {
}
MyVector::MyVector( vector<float> indata ) {
    data = indata;
};
```

This strategy is open to errors because of some amount of code duplication. The better idea is to let the first constructor delegate to the second.

```
// object/delegate.cpp
// delegating constructor
MyVector::MyVector( int nsize )
    : MyVector( vector<float>(nsize) ) {
}
MyVector::MyVector( vector<float> indata ) {
    data = indata;
};
```

### 9.4.2 Copy constructor

Just like the default constructor which is defined if you don't define an explicit constructor, a *copy constructor* is also implicitly defined. There are two ways you can do a copy, and they invoke two slightly different constructors:

```
my_object y(something); // regular or default constructor
my_object x(y);          // copy constructor
my_object x = y;          // copy assignment constructor
```

Usually the copy constructor that is implicitly defined does the right thing: it copies all data members. (If you want to define your own copy constructor, you need to know its prototype. We will not go into that.)

As an example of the copy constructor in action, let's define a class that stores an integer as data member:

```
// object/copyscalar.cpp
class has_int {
private:
    int mine{1};
public:
    has_int(int v) {
        cout << "set: " << v
        << '\n';
        mine = v; };
    has_int( has_int &h ) {
        auto v = h.mine;
        cout << "copy: " << v
        << '\n';
        mine = v; };
    void printme() {
        cout << "I have: " << mine
        << '\n'; };
};
```

The following code shows that the data got copied over:

|                                                                                                                                                                              |                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <b>Code:</b>                                                                                                                                                                 | <b>Output:</b>                                                                       |
| <pre>// object/copyscalar.cpp has_int an_int(5); has_int other_int(an_int); an_int.printme(); other_int.printme(); has_int yet_other = other_int; yet_other.printme();</pre> | <pre>[object] copyscalar: set: 5 copy: 5 I have: 5 I have: 5 copy: 5 I have: 5</pre> |

Class with a vector:

```
// object/copyvector.cpp
class has_vector {
private:
    vector<int> myvector;
public:
```

## 9. Classes and objects

```
has_vector(int v) { myvector.push_back(v); }
void set(int v) { myvector.at(0) = v; }
void printme() { cout
    << "I have: " << myvector.at(0) << '\n'; }
};
```

Copying is recursive, so the copy has its own vector:

**Code:**

```
// object/copyvector.cpp
has_vector a_vector(5);
has_vector other_vector(a_vector);
a_vector.set(3);
a_vector.printme();
other_vector.printme();
```

**Output:**  
[object] copyvector:  
I have: 3  
I have: 5

### 9.4.3 Destructor

Analogous to the constructor routine to create an object, the *destructor* destroys the object. As with the case of the default constructor, there is a default destructor, which you can replace with your own.

A destructor can be useful if your object contains dynamically created data: you want to use the destructor to dispose of that dynamic data to prevent a *memory leak*. Another example is closing files for which the *file handle* is stored in the object.

The destructor is typically called without you noticing it. For instance, any statically created object is destroyed when the control flow leaves its scope.

Example:

**Code:**

```
// object/destructor.cpp
class SomeObject {
public:
    SomeObject() {
        cout << "calling the constructor"
        << '\n';
    }
    ~SomeObject() {
        cout << "calling the destructor"
        << '\n';
    }
};
```

**Output:**  
[object] destructor:  
Before the nested scope  
calling the constructor  
Inside the nested scope  
calling the destructor  
After the nested scope

#### Exercise 9.14. Write a class

```
class HasInt {
private:
    int mydata;
public:
```

```
HasInt(int v) { /* initialize */ };
...
}
```

used as

**Code:**

```
// object/destructexercise.cpp
{
    HasInt v(5);
    v.set(6);
    v.set(-2);
}
```

**Output:**

```
[object] destructexercise:
***** object created with 5 *****
***** object set to 6 *****
***** object set to -2 *****
***** object destroyed after 2
    ↪updates ****
```

The destructor is called when you throw an exception:

**Code:**

```
// object/exceptdestruct.cpp
class SomeObject {
public:
    SomeObject() {
        cout << "calling the constructor"
        << '\n'; }
    ~SomeObject() {
        cout << "calling the destructor"
        << '\n'; }
};

/* ... */
try {
    SomeObject obj;
    cout << "Inside the nested scope" << '\n';
    throw(1);
} catch (...) {
    cout << "Exception caught" << '\n';
}
```

**Output:**

```
[object] exceptdestruct:
calling the constructor
Inside the nested scope
calling the destructor
Exception caught
```

**Remark** A library that does this constructor tracing: <https://github.com/VincentZalzal/noisy>

## 9.5 Advanced topics

### 9.5.1 Static variables and methods

Class members prefixed with `static` behave as if they are not unique to each object of that class, but shared between them.

The standard use for this is to count how many objects of a class have been created. The constructor would increment this static variable and assign it to a private variable:

```
Thing::Thing() {
    mynumber = n_things++; }
```

## 9. Classes and objects

---

Declaring the static variable takes the keywords `static` and `inline`:

```
class Thing {
private:
    static inline int n_things=0; // global count
    int mynumber; // who am I?
```

### 9.5.1.1 Static methods

If you want to query the above static variables you can of course query them from any particular object, since they all have the same value. However, you can define a static method:

```
class Thing {
public:
    static int number_of_things() { return n_things; };
```

and this method can be called on the class itself;

```
cout << "Number of things: " << Thing::number_of_things() << '\n';
```

A static member acts as if it's shared between all objects.

(Note: C++17 syntax)

**Code:**

```
// link/static17.cpp
class myclass {
private:
    static inline int count=0;
public:
    myclass() { ++count; }
    int create_count() {
        return count;
    }
    /* ... */
myclass obj1,obj2;
cout << "I have defined "
    << obj1.create_count()
    << " objects" << '\n';
```

**Output:**

```
[link] static17:
I have defined 2 objects
```

### 9.5.1.2 Legacy syntax for initialization

Prior to C++17, initializing the static variable was done in a strange way. Currently, by adding the keyword `inline`, you can write:

```
Class that counts how many objects have been generated:
```

```
Code:
// object/static.cpp
class Thing {
private:
    static inline int nthings{0};
    int mynumber;
public:
    Thing() {
        mynumber = nthings++;
        cout << "I am thing "
            << mynumber << '\n';
    };
};
```

**Output:**  
[object] static:  
I am thing 0  
I am thing 1  
I am thing 2

(the `inline` is needed for the initialization)

**Exercise 9.15.** If you are doing the prime project (chapter 45), now is a good time to do exercise in section 45.6.1.

In case you come across it in legacy code, C++11 used the following syntax for static class members:

```
// link/static.cpp
class myclass {
private:
    static int count;
public:
    myclass() { ++count; };
    int create_count() { return count; };
};

/* ... */

// in main program
int myclass::count=0;
```

## 9.5.2 Inheriting from standard containers

See section 10.7.

You could write

```
class namedvec {
private:
    std::string name;
    std::vector<float> contents;
public:
    namedvec( std::string n, int s );
    // ...
};
```

The problem now is that for every vector method, `at`, `size`, `push_back`, you have to re-implement that for your `namedvec`.

## 9. Classes and objects

---

Named vector inherits from standard vector:

```
// object/container0.cpp
#include <vector>
#include <string>
class namedvector
    : public std::vector<int> {
private:
    std::string name_;
public:
    namedvector
        ( std::string n, int s )
        : name_(n)
        , std::vector<int>(s) {};
    auto name() {
        return name_; };
};
```

```
// object/container0.cpp
namedvector fivevec("five",5);
cout << fivevec.name()
    << ":" 
    << fivevec.size()
    << '\n';
cout << "at zero: "
    << fivevec.at(0)
    << '\n';
```

**Exercise 9.16.** Extend the code for `namedvector` to make the class templated.

```
// object/container.cpp
namedvector<float> fivetemp("five",5);
cout << fivetemp.name()
    << ":" 
    << fivetemp.size() << '\n';
cout << "at zero: "
    << fivetemp.at(0) << '\n';
```

**Exercise 9.17.** Extend the code from 9.62 and 9.16 to make a namespaced class `geo::vector` that has the functionality of `namedvector`.

```
// object/container.cpp
using namespace geo;
geo::vector<float> float4("four",4);
cout << float4.name() << '\n';
float4[1] = 3.14;
cout << float4.at(1) << '\n';
geo::vector<std::string> string3("three",3);
string3.at(2) = "abc";
cout << string3[2] << '\n';
```

### 9.5.3 Class declarations

For purposes of organizing your code, you may sometimes not want to include the full code of a method, for instance in a header file. This is the distinction between a *declaration* and *definition*.

You have seen this with functions:

```
float f(int);
```

is the *declaration* of a function, stating the name of the function, the types of the input parameters and the type of the return result. (This is sometimes also called the ‘signature’ or ‘prototype’ or ‘header’ of the function.) On the other hand

```
float f(int n) { return sqrt(n); }
```

is the *definition* of the function, giving its full code.

Similarly, you can write class declaration, giving only the data members and signatures of the class methods, and specify the full method later or elsewhere. This can for instance happen if you split your program over multiple files; see chapter 19 and in particular section 19.3.

Declaration:

```
class Point {
private:
    float x,y;
public:
    Point(float x,float y);
    float distance();
};
```

Definition:

```
Point::Point()
    : x(x),y(y) {};
float Point::distance() {
    return sqrt( x*x + y*y );
};
```

- Methods, including constructors, are only given by their function header in the class definition.
- Methods and constructors are then given their full definition elsewhere with ‘classname-double-colon-methodname’ as their name.
- (qualifiers like **const** are given in both places.)

#### 9.5.4 Returning by reference

*Before doing this section, make sure you study section 15.3.*

With all the above discussion of the API abstracting away from internals, sometimes you really want to access the internals of an object directly. The simplest solution is to return a copy:

```
class Foo {
private:
    int x;
public:
    int the_x() { return x; };
};
```

There are two problems with this:

- Returning a copy can be expensive if the internal data is a big object; and
- Maybe you actually want to alter the internal data.

So we have two scenarios:

- you want to get a reference to a data member, in order to alter it;
- you want to get a reference to a data member because copying is expensive, but you will not alter it.

First we show the general mechanism of returning a reference to private data.

Return a reference to a private member:

```
class Point {
private:
    double x,y;
public:
    double &x_component() { return x; };
```

```
};

int main() {
    Point v;
    v.x_component() = 3.1;
}
```

Only define this if you need to be able to alter the internal entity.

Next we show a mechanism that can be considered a performance optimization to this: we return a reference to private data, but in such a way that the calling side can not alter it.

Returning a reference saves you on copying.

Prevent unwanted changes by using a ‘const reference’.

```
class Grid {
private:
    vector<Point> thepoints;
public:
    const vector<Point> &points() const {
        return thepoints; }
};

int main() {
    Grid grid;
    cout << grid.points()[0];
    // grid.points()[0] = whatever ILLEGAL
}
```

### 9.5.5 Accessor functions

It is a good idea to make the data in an object private, so that you can control who has outside access to it.

- Sometimes this private data is auxiliary, and there is no reason for outside access.
- Sometimes you do want outside access, but you want to precisely control how.

Accessor functions:

```
class thing {
private:
    float x;
public:
    float get_x() { return x; }
    void set_x(float v) { x = v; }
```

This has advantages:

- You can print out any time you get/set the value; great for debugging:
 

```
void set_x(float v) {
    cout << "setting: " << v << endl;
    x = v; }
```
- You can catch specific values: if `x` is always supposed to be positive, print an error (throw an exception) if non-positive.

Having two accessors can be a little clumsy. Is it possible to use the same accessor for getting and setting?

Use a single accessor for getting and setting:

**Code:**

```
// object/accessref.cpp
class SomeObject {
private:
    float x=0.;
public:
    SomeObject( float v ) : x(v) {};
    float &xvalue() { return x; };
};

int main() {
    SomeObject myobject(1.);
    cout << "Object member initially :"
        << myobject.xvalue() << '\n';
    myobject.xvalue() = 3.;
    cout << "Object member updated   :"
        << myobject.xvalue() << '\n';
}
```

**Output:**  
**[object] accessref:**

```
Object member initially :1
Object member updated   :3
```

The function `xvalue` returns a reference to the internal variable `x`.

Of course you should only do this if you want the internal variable to be directly changeable!

## 9.5.6 Polymorphism

You can have multiple methods with the same name, as long as they can be distinguished by their argument types. This is known as *polymorphism*; see section [7.7.2](#).

## 9.5.7 Operator overloading

Instead of writing

```
myobject.plus(anotherobject)
```

you can actually redefine the + operator so that

```
myobject + anotherobject
```

is legal. This is known as *operator overloading*: you give your own definition of common arithmetic operators.

Syntax:

```
<returntype> operator<op>( <argument> ) { <definition> }
```

For instance:

## 9. Classes and objects

**Code:**

```
// geom/pointscale.cpp
Point Point::operator*(float f) {
    return Point(f*x, f*y);
}
/* ... */
printf("p1 to origin {:.5}",
       p1.dist_to_origin());
Point scale2r = p1*2.;
printf("scaled right: {}",
       scale2r.dist_to_origin());
// ILLEGAL Point scale2l = 2.*p1;
```

**Output:**

```
[geom] pointmult:
p1 to origin 2.2361
scaled right: 4.472136
```

**Exercise 9.18.** Write a *Fraction* class, and define the arithmetic operators on it.

Define both the + and += operators. Can you use one of them in defining the other?

**Exercise 9.19.** If you know about templates, you can do the exercises in section 22.3.

### 9.5.7.1 Functors

A special case of operator overloading is *overloading the parentheses*. This makes an object look like a function; we call this a *functor*.

Simple example of overloading parentheses:

**Code:**

```
// object/functor.cpp
class IntPrintFunctor {
public:
    void operator()(int x) {
        printf("{}\n", x);
    }
}
/* ... */
IntPrintFunctor intprint;
intprint(5);
```

**Output:**

```
[object] functor:
5
```

**Exercise 9.20.** Extend that class as follows: instead of printing the argument directly, it should print it multiplied by a scalar. That scalar should be set in the constructor. Make the following code work:

**Code:**

```
// object/functor.cpp
IntPrintTimes printx2(2);
printx2(1);
for ( auto i : {5,6,7,8} )
    printx2(i);
```

**Output:**

```
[object] functor2:
2
10
12
14
16
```

### 9.5.7.2 Object outputting

Wouldn't it be nice if you could use `cout` (or `format`) to output a class object?

```
MyObject x;
cout << x << '\n';
```

The reason this doesn't work, is that the 'double less' is a binary operator, which is not defined with your class as second operand.

See section 12.6 for the solution.

### 9.5.8 Constructors and contained classes

Suppose we have a class where each object contains another object of some non-trivial class. Now we have to be aware of how the creation of the outer object relates to that of the inner.

Finally, if a class contains objects of another class,

```
class Inner {
public:
    Inner(int i) { /* ... */ }
};

class Outer {
private:
    Inner contained;
public:
};
```

then

|                                                          |                                                                    |
|----------------------------------------------------------|--------------------------------------------------------------------|
| <pre>Outer( int n ) {     contained = Inner(n); };</pre> | <pre>Outer( int n ) : contained(Inner(n)) {     /* ... */ };</pre> |
|----------------------------------------------------------|--------------------------------------------------------------------|

1. This first calls the default constructor
2. then calls the `Inner(n)` constructor,
3. then copies the result over the `contained` member.

1. This creates the `Inner(n)` object,
2. placed it in the `contained` member,
3. does the rest of the constructor, if any.

**Remark** The order of the member initializer list is ignored: the members specified will be initialized in the order in which they are declared. There are cases where this distinction matters, so best put both in the same order.

### 9.5.9 'this' pointer

Inside an object, a *pointer* to the object is available as `this`. You sometimes come across this in a redundant way, writing `this->x` to refer to a member, where `x` would suffice.

## 9. Classes and objects

---

A pointer to the object itself is available as `this`. Variables of the current object can be accessed this way:

```
class Myclass {
private:
    int myint;
public:
    Myclass(int myint) {
        this->myint = myint;    // option 1
        (*this).myint = myint;  // option 2
    };
};
```

There are legitimate uses of the `this` pointer.

You don't often need the `this` pointer. Example: you need to call a function inside a method that needs the object as argument)

```
/* forward definition: */ class someclass;
void somefunction(const someclass &c) {
    /* ... */
class someclass {
// method:
void somemethod() {
    somefunction(*this);
};

(Rare use of dereference star)
```

There is another interesting idiom that uses the ‘this’ pointer. Define a simple class

```
// objectthis.cpp
class number {
private:
    float x;
public:
    number(float x) : x(x) {};
    float value() { return x; };
```

Defining a method

```
number addcopy(float y) {
    x += y;
    return *this;
};
```

both alters the object, and returns a copy.

Changing the method to return a reference:

```
// objectthis.cpp
number& add(float y) {
    x += y;
    return *this;
};
number& multiply(float y) {
    x *= y;
```

```
    return *this;
};
```

has the interesting effect that no copy is created, but the returned ‘object’ is the object itself, by reference. This makes an interesting idiom possible:

|                                                                                                                                                                                                                |                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| <b>Code:</b>                                                                                                                                                                                                   | <b>Output:</b>                   |
| <pre>// object/this.cpp number mynumber(1.0); mynumber.add(.5); cout &lt;&lt; mynumber.value() &lt;&lt; '\n';  mynumber.multiply(2.).add(1.).multiply(3.); cout &lt;&lt; mynumber.value() &lt;&lt; '\n';</pre> | <pre>[object] this: 1.5 12</pre> |

### 9.5.10 Deducing this

Object methods have access to the object itself through an implicitly defined `this` pointer. In C++23 this pointer can be made explicit:

```
class Foo {
int n;
void f( this Foo& self ) { self.n; };
};
```

The keyword `this` in first position indicates that the first parameter is the object. This has an important application: it eliminates the need for const/non-const overloads of the same function.

```
// object/deducing.cpp
template<typename Self>
auto& x(this Self&& self) {
    return self.x_;
};
```

### 9.5.11 Mutable data

Suppose you have a class and you want to return some complicated data member by const-ref:

```
class has_stuff {
private:
    complicated thing;
public:
    const complicated& get_thing() const {
        return thing;
    };
};
```

To make life interesting, the complicated thing should only be constructed when needed. You could try constructing it in the `get_thing` method:

```
private:
optional<complicated> thing = {};
```

## 9. Classes and objects

---

```
public:
const complicated& get_thing() const {
    if (not thing.has_value()) 
        thing = complicated(/* current stuff */);
    return thing.value(); };
}
```

The problem is that the `get_thing` method now is no longer ‘`const`’. Here you need to realize that `const` means that the routine is only outwardly constant: it can still alter internal data, if that is declared `mutable`:

```
private:
mutable optional<complicated> thing = {};
public:
const complicated& get_thing() const /*as above */
```

|                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                                                                                           |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Code:</b><br><pre>// object/mutable.cpp class has_stuff { private:     <b>mutable optional&lt;complicated&gt;</b> thing = {}; public:     <b>const</b> complicated&amp; get_thing() <b>const</b> {         <b>if</b> (<b>not</b> thing.<b>has_value</b>())              thing = complicated(5);         <b>else cout</b> &lt;&lt; "thing already there\n";         <b>return</b> thing.<b>value</b>();     }; }</pre> | <b>Output:</b><br><b>[object] mutable:</b><br><i>making complicated thing</i><br><i>thing already there</i><br><i>thing already there</i> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|

### 9.5.12 Lazy evaluation

The following idiom prevents creation of temporaries:

```
// gemm/axpy.cpp
template< typename T >
class scaledvector {
public:
    const T& scalar;
    const vector<T>& data;
public:
    scaledvector( T x, const vector<T>& ar )
        :scalar(x), data(ar) {};
    size_t size() const { return data.size();
    () ; };
};

template< typename T >
scaledvector<T> operator*( T x, const
    vector<T>& ar ) {
    return scaledvector<T>(x, ar);
};

template< typename T >
vector<T>& operator+=( vector<T>& y, const
    scaledvector<T>& ax ) {
    assert( y.size() == ax.size() );
    for ( size_t i=0; i < y.size(); ++i )
        y[i] += ax.scalar * ax.data[i];
    return y;
};
```

## 9.6 Review questions

**Review 9.2.** Fill in the missing term

- The functionality of a class is determined by its...
- The state of an object is determined by its...

How many constructors do you need to specify in a class definition?

- Zero
- Zero or more
- One
- One or more

**Review 9.3.** Describe various ways to initialize the members of an object.



# Chapter 10

## Arrays

An array<sup>1</sup> is an indexed data structure that for each index stores an integer, floating point number, character, object, et cetera. In scientific applications, arrays often correspond to vectors and matrices, potentially of quite large size. (If you know about the Finite Element Method (FEM), you know that vectors can have sizes in the millions or beyond.)

In this chapter you will see the C++ `vector` construct, which implements the notion of an array of things, whether they be numbers, strings, objects.

*C difference:* Avoid automatic arrays. While C++ can use the C mechanisms for arrays, for almost all purposes it is better to use `vector`. In particular, this is a safer way to do dynamic allocation. The old mechanisms are briefly discussed in section 10.10.

### 10.1 Some simple examples

#### 10.1.1 Vector creation

To use vectors, you first need the `vector` header from the standard library. This allows you to declare a vector, specifying what type of element it contains. Next you may want to decide how many elements it contains; you can specify this when you declare the vector, or determine it later, dynamically.

We start with the most obvious way of creating a vector: enumerating its elements.

Short vectors can be created by enumerating their elements:

```
// array/shortvector.cpp
#include <vector>
using std::vector;

int main() {
    vector<int> evens{0, 2, 4, 6, 8};
    vector<float> halves = {0.5, 1.5, 2.5};
    auto halffloats = {0.5f, 1.5f, 2.5f};
    cout << evens.at(0)
        << " from " << evens.size()
        << '\n';
    return 0;
```

---

1. The term ‘array’ is used informally here. The `array` keyword, is briefly discussed in section 10.4.

```
}
```

**Exercise 10.1.**

1. Take the above snippet, compile, run.
2. Add a statement that alters the value of a vector element. Check that it does what you think it does.
3. Add a vector of the same length as the `evens` vector, containing odd numbers which are the even values plus 1?

You can base this off the file `shortvector.cpp` in the repository

For a less simple example, let's make a `vector` containing objects, in this case the `Point` objects:

```
vector<Point> diagonal =
{ {0.,0.}, {1.,1.}, {1.5,1.5}, {2.,2.}, {3.,3.} };
```

### 10.1.2 Initialization

There are various ways to declare a vector, and possibly initialize it.

More generally, vectors can be defined

- Without further specification, creating an empty vector:

```
vector<float> some_numbers;
```

- With a size indicated, allocating that number of elements:

```
vector<float> five_numbers(5);
```

(This sets the elements to a default value; zero for numeric types.)

- You can initialize a vector with a constant:

```
vector<float> x(25, 3.15);
```

which defines a vector `x` of size 25, with all elements initialized to 3.15.

If your vector is short enough, you can set all elements explicitly with an *initializer list*, as you saw above. Note that the size is not specified here, but is deduced from the length of the initializer list:

**Code :**

```
// array/dynamicinit.cpp
{
    vector<int> numbers{5, 6, 7, 8, 9, 10};
    cout << numbers.at(3) << '\n';
}
{
    vector<int> numbers = {5, 6, 7, 8, 9, 10};
    numbers.at(3) = 21;
    cout << numbers.at(3) << '\n';
}
```

**Output :**

```
[array] dynamicinit:
```

```
8
```

```
21
```

**Review 10.1.** T/F?

- It is possible to write a valid C++ program where you define a variable `vector`.

**10.1.3 Element access**

There are two ways of accessing vector elements.

1. With the ‘dot’ notation that you know from structures and objects, you can use the `at` method:

**Code:**

```
// array/assign.cpp
vector<int> numbers = {1, 4};
numbers.at(0) += 3;
numbers.at(1) = 8;
cout << numbers.at(0) << ","
    << numbers.at(1) << '\n';
```

**Output:**  
[array] assignatfun:  
4, 8

The expression `a.at(i)` can be used to get the value of a vector element, or it can occur in the left-hand side of an assignment to set the value.

2. Alternatively, use the same notation as in C):

**Code:**

```
// array/assign.cpp
vector<int> numbers = {1, 4};
numbers[0] += 3;
numbers[1] = 8;
cout << numbers[0] << ","
    << numbers[1] << '\n';
```

**Output:**  
[array] assignbracket:  
4, 8

Again, the element accessed can be used both in left and right hand sides.

Indexing starts at zero. Consequently, a vector declared as

```
vector<int> ints(N)
```

has elements  $0, \dots, N - 1$ .

**10.1.4 Access out of bounds**

Have you wondered what happens if you access a vector element outside the bounds of the vector?

```
vector<float> x(6); // size 6, index ranges 0..5
x.at(6) = 5.; // oops!
i = -2;
x[i] = 3; // also oops, but different.
```

Often it is hard for the compiler to determine that you are accessing an element outside the vector bounds. Most likely, it will only be detected at runtime. The two access methods now differ in how they do *vector bounds checking*.

1. Using the `at` method will always do a bounds test, and exit your program immediately if you access an element outside the vector bounds. (Technically, it throws an *exception*; see section 23.2 for how this works and how you can handle this.)
2. The bracket notation `a[i]` performs no bounds tests: it calculates a memory address based on the vector location and the index, and attempts to return what is there. As you may imagine, this lack of checking makes your code a little faster. However, it also makes your code unsafe:
  - Your program may crash with a *segmentation fault* or *bus error*, but no clear indication where and why this happened. (Such a crash can be caused by other things than vector access out of bounds.)
  - Your program may continue running, but giving wrong results, since reading from outside the vector probably gives you meaningless values. Writing outside the bounds of an vector may even change the data of other variables, leading to really strange errors.

For now, it is best to use the `at` method throughout.

Indexing out of bounds can go undetected for a while:

|                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                        |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Code :</b><br><pre>// array/segmentation.cpp vector&lt;float&gt; v(10, 2); for (int i=5; i&lt;6; i--)     cout &lt;&lt; "element " &lt;&lt; i         &lt;&lt; " is " &lt;&lt; v[i] &lt;&lt; '\n';</pre> | <b>Output :</b><br><b>[array] segmentation:</b><br><pre>element -5869 is 0 element -5870 is 2.8026e-45 element -5871 is 2.38221e-43 element -5872 is 1.00893e-41 element -5873 is 0 element -5874 is 0 element -5875 is 0 element -5876 is 0 /bin/sh: line 1: 48082   ↳Segmentation fault: 11   ↳(core dumped)   ↳./segmentation</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Exercise 10.2.** The following codes are not correct in some sense. How will this manifest itself?

```
// loop/vecerr.cpp
vector<int> a(5);
a[6] = 1.;
cout << "Success\n";

// loop/vecexc.cpp
vector<int> a(5);
a.at(6) = 1.;
```

## 10.2 Going over all vector elements

If you need to consider all the elements in a vector, you typically use a `for` loop. There are various ways of doing this.

Conceptually, a `vector` can correspond to a set of things, and the fact that they are indexed is purely incidental, or it can correspond to an ordered set, and the index is essential. If your algorithm requires you to access all elements, it is important to think about which of these cases apply, since there are two different mechanism.

### 10.2.1 Ranging over a vector

First of all consider the cases where you consider the vector as a collection of elements, and the loop functions like a mathematical ‘for all’. This uses a ‘colon’ notation.

A range-based `for` loop gives you directly the element values:

```
vector<float> my_data(N);
/* set the elements somehow */;
for ( float e : my_data )
    // statement about element e
```

Here there are no indices because you don’t need them.

You can spell out the type of the vector element, but such type specifications can be complex. In that case, using *type deduction* through the `auto` keyword is quite convenient.

Same with `auto` instead of an explicit type for the elements:

```
for ( auto e : my_data )
    // same, with type deduced by compiler
```

As an example, consider finding the maximum value in an array of numbers. Since we only want the value, not where it is placed, we use this range-based syntax. (Note: there is actually a library function for this, so this example is mostly for the sake of discussing the ranging mechanism.)

Finding the maximum element

Code:

```
// array/dynamicmax.cpp
vector<int> numbers = {1, 4, 2, 6, 5};
int tmp_max = -2000000000;
for (auto v : numbers)
    if (v > tmp_max)
        tmp_max = v;
cout << "Max: " << tmp_max
    << " (should be 6)" << '\n';
```

Output:

```
[array] dynamicmax:
Max: 6 (should be 6)
```

Another note: the initialization to a large negative value can be done more elegantly; see section 24.2.

So-called *initializer lists* can also be used as a list denotation:

```
Code:
// array/rangedenote.cpp
for ( auto i : {2,3,5,7,9} )
    cout << i << ",";
cout << '\n';
```

```
Output:
[array] rangedenote:
2,3,5,7,9,
```

### 10.2.2 Ranging over the indices

If you actually need the index of the element, you can use a traditional `for` loop with loop variable.

You can write an *indexed for* loop, which uses an index variable that ranges from the first to the last element.

```
for (int i= /* from first to last index */ )
    // statement about index i
```

Example: find the maximum element in the vector, and where it occurs.

```
Code:
// array/vectoridxmax.cpp
int tmp_idx = 0;
int tmp_max = numbers.at(tmp_idx);
for (int i=0; i<numbers.size(); ++i) {
    int v = numbers.at(i);
    if (v>tmp_max) {
        tmp_max = v; tmp_idx = i;
    }
}
cout << "Max: " << tmp_max
     << " at index: " << tmp_idx << '\n';
```

```
Output:
[array] vecidxmax:
Max: 6.6 at index: 3
```

**Exercise 10.3.** Indicate for each of the following vector operations whether you prefer to use an indexed loop or a range-based loop. Give a short motivation.

- Count how many elements of a vector are zero.
- Find the location of the last zero.

**Exercise 10.4.** Find the element with maximum absolute value in a vector. Use:

```
vector<int> numbers = {1,-4,2,-6,5};
```

Hint:

```
#include <cmath>
..
absx = abs(x);
```

**Exercise 10.5.** Find the location of the first negative element in a vector.

Which mechanism do you use?

**Exercise 10.6.** Check whether a vector is sorted.

**Remark** In C++20 you can a loop index in a range-based loop, through an initializer statement:

**Code:**

```
// range/enumerate.cpp
vector<int> values{2, 4, 5, 7, 10};
for (size_t i=0; auto v : values)
    cout << "Element " << i++
        << ":" << v << '\n';
```

**Output:**

[range] enumerate:  
Element 0: 2  
Element 1: 4  
Element 2: 5  
Element 3: 7  
Element 4: 10

Note that you have to increment the loop variable explicitly in the loop body.

### 10.2.3 Ranging by reference

In the previous examples we only read out the values of the vector elements. What if you want to access the element values in order to change them, for instance to add something to them, or multiply them?

Range-based loops can do this. You need to realize that in the loops so far

```
for (auto e : my_array)
    // something with e
```

the variable `e` actually contains a copy of the vector elements. This means that altering `e` does not affect the vector. To get that effect, you need to make `e` a reference to the vector elements.

```
for (auto &e : my_vector)
    e = ...
```

Example: multiply all elements by two:

**Code:**

```
// array/vectorrangeref.cpp
vector<float> myvector
    = {1.1, 2.2, 3.3};
for (auto &e : myvector)
    e *= 2;
cout << myvector.at(2)
    << '\n';
```

**Output:**

[array] vectorrangeref:  
6.6

**Exercise 10.7.** If you do the prime numbers project, you can now do exercise 45.19.

#### 10.2.4 Arrays and while loops

In a `while` loop, if you need an index, you need to maintain that index explicitly. There are then certain common idioms.

**Code :**

```
// loop/plusplus.cpp
vector<int> numbers{3, 5, 7, 8, 9, 11};
int index{0};
while (numbers[index] % 2 == 1) ;
cout << "The first even number\n"
    << "appears at index "
    << index << '\n';
```

**Output :**

```
[loop] plusplus:
The first even number
appears at index 4
```

**Exercise 10.8.** Exercise: modify the preceding code so that after the while loop `index` is the number of leading odd elements.

### 10.3 Vector are a class

Above, you created vectors and used functions `at` and `size` on them. They used the dot-notation of class methods, and in fact vector form a `vector` class. You can have a vector of ints, floats, doubles, et cetera; the angle bracket notation indicates what the specific type stored in the vector is. You could say that the vector class is parameterized with the type (see chapter 22 for the details). You could also say that `vector<int>` is a new data type, pronounced ‘vector-of-int’, and you can make variables of that type.

Vectors can be copied just like other datatypes:

**Code :**

```
// array/vectorcopy.cpp
vector<float> v(5, 0), vcopy;
v.at(2) = 3.5;
vcopy = v;
vcopy.at(2) *= 2;
cout << v.at(2) << ","
    << vcopy.at(2) << '\n';
```

**Output :**

```
[array] vectorcopy:
3.5, 7
```

#### 10.3.1 Vector methods

There are several *methods* to the `vector` class. Some of the simpler ones are:

- `at`: index an element
- `size`: give the size of the vector
- `front`: value of the first element
- `back`: value of the last element

There are also methods relating to dynamic storage management, which we will get to next.

**Exercise 10.9.** Create a `vector`  $x$  of `float` elements, and set them to random values. (Use the C random number generator for now.)

Now normalize the vector in  $L_2$  norm and check the correctness of your calculation, that is,

1. Compute the  $L_2$  norm of the vector:

$$\|v\| \equiv \sqrt{\sum_i v_i^2}$$

2. Divide each element by that norm;
3. The norm of the scaled vector should now be 1. Check this.
4. Bonus: your program may be printing 1, but is it actually 1? Investigate.

What type of loop are you using?

### 10.3.2 Vectors are dynamic

A vector can be grown or shrunk after its creation. For instance, you can use the `push_back` method to add elements at the end.

Extend a vector's size with `push_back`:

**Code:**

```
// array/vectorend.cpp
vector<int> mydata(5, 2);
// last element:
cout << mydata.back()
    << '\n';
mydata.push_back(35);
cout << mydata.size()
    << '\n';
// last element:
cout << mydata.back()
    << '\n';
```

**Output:**  
[array] vectorend:  
6  
35

Similar functions: `pop_back`, `insert`, `erase`. Flexibility comes with a price.

It is tempting to use `push_back` to create a vector dynamically.

Known vector size:

```
int n = get_inputsize();
vector<float> data(n);
for ( int i=0; i<n; i++ ) {
    auto x = get_item(i);
    data.at(i) = x;
}
```

Unknown vector size:

```
vector<float> data;
float x;
while ( next_item(x) ) {
    data.push_back(x);
}
```

If you have a guess as to size: `data.reserve(n)`.

(Issue with array-of-object: in left code, constructors are called twice.)

```
vector<int> iarray;
```

creates a vector of size zero. You can then

```
iarray.push_back(5);
iarray.push_back(32);
iarray.push_back(4);
```

However, this dynamic resizing involves memory management, and maybe operating system functions. This will probably be inefficient. Therefore you should use such dynamic mechanisms only when strictly necessary. If you know the size, create a vector with that size. If the size is not precisely known but you have a reasonable upper bound, you can call `reserve` to reserve space for that many elements:

```
vector<int> iarray;
iarray.reserve(100);
while ( ... )
    iarray.push_back( ... );
```

The combination of using `reserve` and `push_back` can be preferable over creating the vector immediately with a certain size. Writing `vector<x> xs(100)`, where `x` is some object, causes the default constructor of `x` to be called on each vector element. For complicated objects this may not be advisable.

## 10.4 The Array class

In scientific computing applications, arrays are unlikely to be extended by `push_back` or shrunk by `erase`: they have a size that is set based on some input parameter and never changed. To express this, you could write your own storage container where such functionality is missing or disabled. For the more specialized case where the size is known at compile-time, use the `array` class.

```
// array/stdarray.cpp
#include <array>
using std::array;
```

Array objects are declared with a static size as:

```
array<float,3> coordinate;

// array/stdarray.cpp
{
    array<float,5> v5;
    cout << "size: " << v5.size() << '\n';
    // WRONG: no such function
    // v5.push_back(2);
}
```

### 10.4.1 Initialization

There are several ways to initialize a `std::array`.

With template parameters:

```
array<int,3> i3 = {1,2,3};  
// or  
array<int,3> i3 { {1,2,3} } ;  
// or  
array<int,3> i3{1,2,3};
```

With C++17 *template argument deduction*:

```
array i3 = {1,2,3};  
// Note: DOES NOT COMPILE:  
array not4{1.5,2,3,4};
```

### 10.4.2 Pass as argument

Passing an *initializer list* to a function that has a `std::array` parameter works:

```
// array/toarray.cpp  
float first_of_three( array<float,3> fff ) {  
    return fff[0];  
};  
/* ... */  
println("{ }",  
    first_of_three( { 1.5 , 2.5 , 3.5 } ) );
```

However, in some cases it doesn't:

```
// array/toarray.cpp  
template<unsigned long int d>  
float first_of_bunch( array<float,d> fff ) {  
    return fff[0];  
};  
/* ... */  
// DOES NOT COMPILE:  
// cout << first_of_bunch( { 1.5f , 2.5f , 3.5f } ) << '\n';  
// couldn't deduce template parameter 'd'
```

For this we can use `to_array` in C++20:

```
// array/toarray.cpp  
// template parameter 'd' can be deduced from 'to_array':  
println("{ }",  
    first_of_bunch( to_array( { 1.5f , 2.5f , 3.5f } ) ) );
```

## 10.5 Vectors and functions

Vectors act like any other datatype, so they can be used with functions: you can pass a vector as argument, or have it as return type. We will explore that in this section.

### 10.5.1 Pass vector to function

The mechanisms of parameters passing (section 7.5) apply to vectors too: they can be passed by value and by reference.

First of all, you can pass by value; section 7.5.1. Here, the vector argument is copied to the function; the function receives a full copy of the vector, and any changes to that vector in the function do not affect the calling environment.

## 10. Arrays

---

*C difference:* C++ vectors and C arrays differ in how they are passed to functions. In C the array is not copied: you pass the address by value. Not the contents.

**Code:**

```
// array/vectorpassnot.cpp
void set0
    ( vector<float> v, float x )
{
    v.at(0) = x;
}
/* ... */
vector<float> v(1);
v.at(0) = 3.5;
set0(v, 4.6);
cout << v.at(0) << '\n';
```

**Output:**  
[array] vectorpassnot:  
3.5

- Vector is copied
- ‘Original’ in the calling environment not affected
- Cost of copying?

**Exercise 10.10.** Revisit exercise 10.9 and introduce a function for computing the  $L_2$  norm.

Next, you can pass by reference; section 7.5.2. Here, the parameter vector becomes alias to the vector in the calling environment, so changes to the vector in the function affect the argument vector in the calling environment.

**Code:**

```
// array/vectorpassref.cpp
void set0
    ( vector<float> &v, float x )
{
    v.at(0) = x;
}
/* ... */
vector<float> v(1);
v.at(0) = 3.5;
set0(v, 4.6);
cout << v.at(0) << '\n';
```

**Output:**  
[array] vectorpassref:  
4.6

An important reason for wanting to pass by reference is that it avoids the possibly substantial cost in copying the argument in passing by value. So what if you want that efficiency, but you like to safeguard yourself against inadvertent changes to the argument vector? For this, you can declare the function parameter as ‘const reference’.

Passing a vector that does not need to be altered:

```
int f( const vector<int> &ivec ) { ... }
```

- Zero copying cost
- Not alterable, so: safe!
- (No need for pointers!)

The general guideline for parameter passing was

- pass by value if the argument is not altered;
- pass by reference if the argument is altered.

For vectors this matter gets another dimension: passing by value means copying, which is potentially expensive for vectors. The way out here is to pass by *const reference* which both prevents copying and prevents accidental altering; see section 18.2.

### 10.5.2 Vector as function return

You can have a vector as return type of a function.

Example: this function creates a vector, with the first element set to the size:

**Code:**

```
// array/vectorreturn.cpp
vector<int> make_vector(int n) {
    vector<int> x(n);
    x.at(0) = n;
    return x;
}
/* ... */
vector<int> x1 = make_vector(10);
// "auto" also possible!
cout << "x1 size: "
     << x1.size() << '\n';
cout << "zero element check: "
     << x1.at(0) << '\n';
```

**Output:**

```
[array] vectorreturn:
x1 size: 10
zero element check: 10
```

**Exercise 10.11.** Write a function of one `int` argument *n*, which returns vector of length *n*, and which contains the first *n* squares.

**Exercise 10.12.** Write functions `random_vector` and `sort` to make the following main program work:

```
int length = 10;
vector<float> values = random_vector(length);
vector<float> sorted = sort(values);
```

This creates a vector of random values of a specified length, and then makes a sorted copy of it.

Instead of making a sorted copy, sort in-place  
(overwrite original data with sorted data):

```
int length = 10;
vector<float> values = random_vector(length);
sort(values); // the vector is now sorted
```

Find arguments for/against that approach.

(Note: C++ has sorting functions built in.)

(See section [24.7.5](#) for the random function.)

**Exercise 10.13.** Write code to take a vector of integers, and construct two vectors, one containing all the odd inputs, and one containing all the even inputs. So:

```
input:  
 5, 6, 2, 4, 5  
output:  
 5, 5  
 6, 2, 4
```

Can you write a function that accepts a vector and produces two vectors as described?

## 10.6 Vectors in classes

You may want a class of objects that contain a vector. For instance, you may want to name your vectors.

```
class named_field {  
private:  
    vector<double> values;  
    string name;
```

The problem here is when and how that vector is going to be created.

- If the size of the vector is statically determined, you can of course declare it with that size:

```
class named_field {  
private:  
    vector<double> values(25);  
    string name;
```

- ... but in the more interesting case the size is determined during the runtime of the program. In that case you would declare:

```
named_field velocity_field(25, "velocity");
```

specifying the size in the constructor of the object.

So now the question is, how do you allocate that vector in the object constructor?

One solution would be to specify a vector without size in the class definition, create a vector in the constructor, and assign that to the vector member of the object:

```
named_field( int n ) {  
    values = vector<int>(n);  
}
```

However, this has the effect that

- The constructor first creates `values` as a zero size vector,
- then it creates an anonymous vector of size `n`,
- and by assigning it, destroys the earlier created zero size vector.

This is somewhat inefficient, and the optimal solution is to create the vector as part of the *member initializer* list:

Use initializers for creating the contained vector:

```
class named_field {
private:
    string name;
    vector<double> values;
public:
    named_field( string name, int n )
        : name(name),
          values(vector<double>(n)) {
    };
};
```

Even shorter:

```
named_field( string name, int n )
    : name(name), values(n) {
```

### 10.6.1 Timing

Different ways of accessing a vector can have drastically different timing cost.

You can push elements into a vector:

```
// array/arraytime.cpp
vector<int> flex;
/* ... */
for (int i=0; i<LENGTH; ++i)
    flex.push_back(i);
```

If you allocate the vector statically, you can assign with `at`:

```
// array/arraytime.cpp
vector<int> stat(LENGTH);
/* ... */
for (int i=0; i<LENGTH; ++i)
    stat.at(i) = i;
```

With subscript:

```
// array/arraytime.cpp
vector<int> stat(LENGTH);
/* ... */
for (int i=0; i<LENGTH; ++i)
    stat[i] = i;
```

You can also use `new` to allocate\*:

```
// array/arraytime.cpp
int *stat = new int[LENGTH];
```

```
/* ... */
for (int i=0; i<LENGTH; ++i)
    stat[i] = i;
```

\* Considered bad practice. Do not use.

For `new`, see section 17.6.2. However, note that this mode of allocation is basically never needed.

Timings are partly predictable, partly surprising:

```
Flexible time: 2.445
Static at time: 1.177
Static assign time: 0.334
Static assign time to new: 0.467
```

The increased time for `new` is a mystery.

So do you use `at` for safety or `[]` for speed? Well, you could use `at` during development of the code, and insert

```
#define at(x) operator[](x)
```

for production.

## 10.7 Wrapping a vector in an object

Sometimes an existing container is almost-but-not-quite good enough.

As a simple example, consider a ‘named vector’, which is a `std::vector`, except that it has a `string` as name. We could realize this as a class that contains both a vector and a string.

```
// array/arrayprint.cpp
class namedvector {
private:
    string name;
    vector<int> values;
public:
    namedvector(int n, string name="unnamed")
        : name(name), values(n) {
    };
    string rendered() {
        stringstream render;
        render << name << ":";
        for (auto v : values)
            render << " " << v << ",";
        return render.str();
    }
    /* ... */
};
```

A problem with this approach is that you may have to recreate some methods to access the vector. For instance, you need to define the `at` method on the object to access elements of the vector:

```
// array/arrayprint.cpp
int &namedvector::at(int i) {
    return values.at(i);
}
```

### 10.7.1 Public inheritance

```
// object/isavector.cpp
class witharray : public vector<float> {
```

Now the constructor calls the constructor of the vector:

```
// object/isavector.cpp
witharray::witharray( float n )
    : vector<float>(n) {
```

but after that the object has all the methods of the vector:

|                                                                                                       |                                     |
|-------------------------------------------------------------------------------------------------------|-------------------------------------|
| <b>Code:</b>                                                                                          | <b>Output:</b>                      |
| <pre>// object/isavector.cpp witharray x(5); x[ x.size()-1 ] = 3.14; println( "{}", x.back() );</pre> | <pre>[object] isavector: 3.14</pre> |

## 10.8 Multi-dimensional cases

C++ traditionally had little native support for multi-dimensional arrays, which are essential for linear algebra operations and many other physics algorithms. In section 63.3 we will look at the *Eigen* library which provides linear algebra objects and operations, but for now we take a look at some of the modern mechanisms that can be used to handle multi-dimensional objects natively in C++.

### 10.8.1 Matrix as vector of vectors

A first – and, spoiler: inferior – way of emulating a multi-dimensional structure is to create a vector-of-vectors:

```
vector<float> row(20);
vector<vector<float>> matrix(10, row);
/* you can now write: */
... matrix[i][j] ...
```

Here you first create a vector that stands for a matrix row, then fill a second vector with a number of copies of that.

To understand the problem with this, we need to go into a little detail on how vectors are realized. You can think of a `std::vector` as consisting of two parts: a small block (allocated on the *stack*) with the size and capacity of the vector, and a block (on the *heap*) with the actual data; see figure 10.1. This is in contrast to C and Fortran, where a vector is only the data array.

Now if you make a vector-of-vectors (figure 10.2) you get a bunch of vectors, potentially spread through-

## 10. Arrays

---

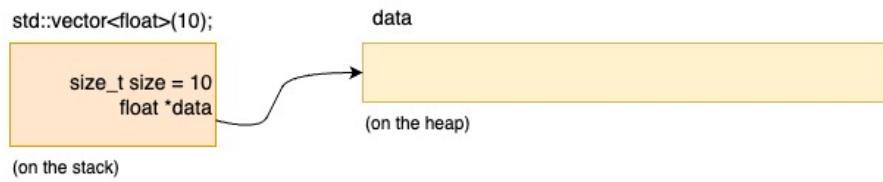


Figure 10.1: Implementation of a standard vector

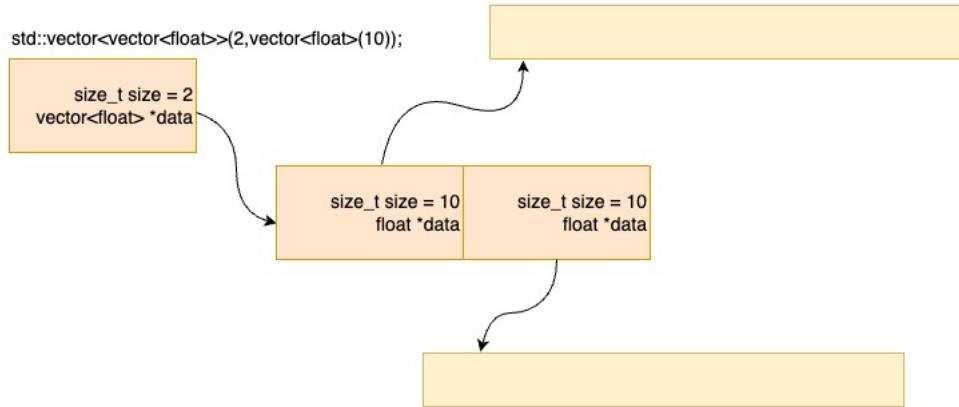


Figure 10.2: A vector of vectors

out memory.

**Remark** You could also have written the above fragment as:

```
vector<vector<float>> matrix(10);
for ( auto &row : matrix ) {
    row = vector<float>(20);
}
```

This loop formulation makes some special effects possible. For instance, can you think how to create a triangular array?

This is not the best implementation of a matrix, for instance because the elements are not contiguous. However, let's continue with it for a moment and write a matrix class. The first thing we need is element access.

```
// array/matrixclass.cpp
class matrix {
private:
    vector<double> matrix_data;
    int m, n;
public:
    matrix(int m, int n)
        : m(m), n(n), matrix_data(m*n) {};
```

(Can you combine the `get`/`set` methods, using 18.4?)

**Exercise 10.14.** Write `rows()` and `cols()` methods for this class that return the number of rows and columns respectively.

**Exercise 10.15.** Write a method `void set(double)` that sets all matrix elements to the same value.

Write a method `double totalsum()` that returns the sum of all elements.

**Code:**

```
// array/matrix.cpp
A.set(3.);
cout << "Sum of elements: "
    << A.totalsum() << '\n';
```

**Output:**

```
[array] matrixsum:
Sum of elements: 30
```

You can base this off the file `matrix.cpp` in the repository

Having these simple access methods, we can start implementing some linear algebra operations.

**Exercise 10.16.** Add methods such as `transpose`, `scale` to your matrix class.

Implement matrix-matrix multiplication.

## 10.8.2 A better matrix class

The problem of the non-contiguous matrix rows can be solved by making a matrix class where the objects store a long enough `vector`. You then need a translation from two-dimensional  $i, j$  indexing to a linear scheme.

Linearized indexing:

Class:

```
// array/matrixclass.cpp
class matrix {
private:
    vector<double> matrix_data;
    int m, n;
public:
    matrix(int m, int n)
        : m(m), n(n), matrix_data(m*n) {};
```

Methods:

```
void setij(int i, int j, double v) {
    matrix_data.at(i*n + j) = v;
}
double getij(int i, int j) {
    return matrix_data.at(i*n + j);
};
```

**Exercise 10.17.** In the matrix class of the previous slide, why are  $m, n$  stored explicitly while that would not be needed in the scheme of section 10.8.1?

One big advantage of having the matrix storage be contiguous is that it is compatible with the storage traditionally used in many libraries and codes. Also, it makes operations more efficient, but to understand that you need to know more about computer architecture.

The syntax for `setij`/`getij` can be improved. (Make sure to study section 18.4 first.)

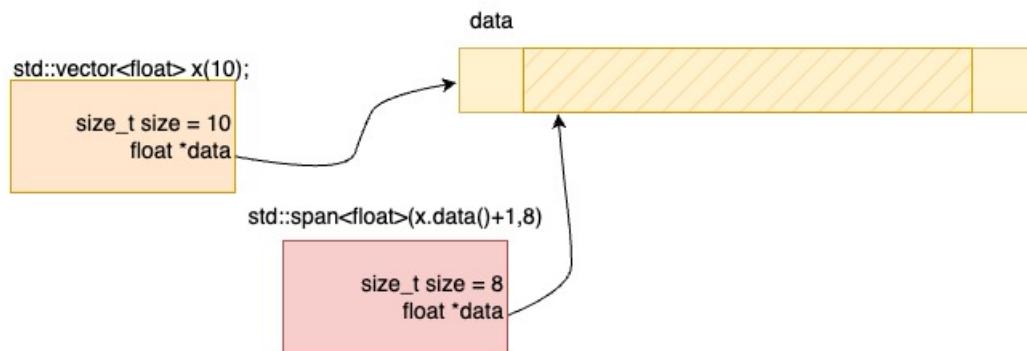


Figure 10.3: Span using pointer into vector data

**Exercise 10.18.** Write a method `element` of type `double&`, so that you can write

```
A.element(2, 3) = 7.24;
```

**Remark** In C++23 it is possible to make the multi-dimensional indexing syntax

```
A[2, 3] = 7.24;
```

work; see section 9.5.7. This is used in `mdspan`; see section 10.8.3.1.

### 10.8.3 Span and mdspan

Arrays in C allowed for some operations that are harder to do with C++ vectors. For instance, you could create a subset of an array, operate on it as if it were an array by itself, and have the original array be affected. Here is an illustration of this mechanism at work, using explicitly allocated data.

Form `subdata` as part of the array `data_array`, starting at the second element:

```
double *data_array = new double[N];
double *subdata = data_array + 1; // pointer arithmetic: point at next element
subdata[1] = 5.; // same as: data_array[2] = 5.;
```

Using ‘subarrays’ would be useful, for instance in a *quicksort* algorithm:

```
// Warning: this is pseudo-code
void qs( data ) {
    if (data.size() > 1) {
        // pivoting stuff omitted
        qs( data.lefthalf() ); qs( data.righthalf() );
    }
}
```

You can not get this same effect with a standard `vector` or `array` since they exclusively own their data. Creating a vector from elements of another vector, for instance with iterators, leads to a copy. The scenario of having non-owned sub-arrays is addressed by the `span` class.

If you really want vector-like objects to share data use `span` class, added in C++20. Since a `span` is created from a pointer and a size, it allows you to create a non-owning view into a `vector`; see figure 10.3.

Create a `span` from a `vector`, starting at its second element and ending before its last:

```
#include <span>
vector<double> v;
std::span<double> v_span( v.data() + 1, v.size() - 2 );
```

The `span` object has the same `data`, and `size` methods, as well as the subscript operator, as a `vector`. Also you can iterate over it as you'll see below, but it has no dynamic methods such as `push_back`. Bound-checked indexing through the `at` method is planned for C++26.

Here is an example of `span` in use: we create a span from part of a `vector`, change an element in the span, and see that the corresponding element in the original vector is changed:

**Code:**

```
// span/subspan.cpp
vector v{1, 2, 3};
span tail( v.data() + 1, 2 );
tail[0] = tail[1] = 0;
println("{} , {} , {}" , v[0], v[1], v[2]);
```

**Output:**

```
[span] subspan1:
1, 0, 3
```

The previous example can be made more elegant by using `last` to create a `subspan`.

Alter a subset of a vector through a subspan:

**Code:**

```
// span/subspan.cpp
vector v{1, 2, 3};
span data( v.data(), v.size() );
span tail = data.last( 2 );
for ( auto& e : tail )
    e = 0;
println("{} , {} , {}" , v[0], v[1], v[2]);
```

**Output:**

```
[span] subspan2:
1, 0, 0
```

### 10.8.3.1 `mdspan`

The C++23 standard introduced `mdspan`, a multi-dimensional span, which is like `span` in the sense of it being a non-owning container-like object. Its big improvement over `span` is that it offers functionality for multi-dimensional indexing. (A benchmark study [18] shows that this incurs no performance penalty over linearized indexing as in section 10.8.2.)

**Remark** The `mdspan` library is not yet implemented in all compilers. The examples in this book have been tested using the reference implementation, available from the Kokkos project. To include this, use:

```
// mdspan/index2.cpp
#include "mdspan/mdspan.hpp"
namespace md = Kokkos;
namespace KokkosEx = MDSPAN_IMPL_STANDARD_NAMESPACE::MDSPAN_IMPL_PROPOSED_NAMESPACE;
namespace mdx = KokkosEx;
```

Consequently, the examples may use `md::mdspan`, which will read `std::mdspan` if your compiler supports it.

An `mdspan` object is a multi-dimensional Cartesian brick, where each dimension can be specified in a number of ways. While `mdspan` can handle ‘dynamic extents’, a common case in scientific computing has known extents in all dimensions. The definition is then fairly simple:

Create 2D `mdspan` from vector:

```
// mdspan/index2std.cpp
// matrix in row major
vector<float> A(M*N);
std::mdspan
Amd{ A.data(), std::extents{M, N} };
```

The `mdspan` class is templated over the basic type, the extents, and the layout. These are either deduced by the compiler using **CTAD!** (**CTAD!**), or have certain defaults. For instance, while the default template parameter value `LayoutPolicy = std::layout_right` correspond to *row-major* storage, (see section 10.8.2), it is also possible to have *column-major* ordering. The latter is the scheme used in *Fortran* and the Basic Linear Algebra Subprograms (BLAS) standard for linear algebra, so you may need it when interfacing across languages.

Declaration with template values specified and using column-major ordering:

```
// mdspan/index2.cpp
// matrix in column major
vector<float> B(M*N);
md::mdspan<float, md::extents<int, M, N>, md::layout_left>
Bmd{ B.data() };
```

## 10.9 Advanced topics

### 10.9.1 The size of a vector

You have seen that you can create a vector with a specified size, or set the size dynamically by adding elements.

The dynamic strategy can be more costly, because the underlying storage may need to be re-allocated. However, this process is done efficiently enough that it amounts to a fixed cost per added element.

### 10.9.2 Loop index type

In indexed loops you may be used to using `int` as the type of the loop variable:

```
for ( int i=0; i<some_array.size(); ++i ) { /* stuff */ }
```

The problem with this is that integers typically have a maximum value of about 2 billion (see section 24.2 for details), and containers such as `vector` can store more elements than that.

First of all, in cases where you strictly operate on array elements you can dispense with the loop variable by using *range-based loops*:

---

```
for ( auto &x : my_vector )
    x *= 2;
```

or standard algorithms:

```
for_each( my_vector, [] ( auto &x ) { x*= 2; } );
```

If you absolutely need that index variable, give it a type of `size_t`:

```
for ( size_t i=0; i<some_array.size(); ++i ) { /* stuff */ }
```

**Remark** Some compilers will indeed issue a warning on the first loop type, but that is not related to the choice of integer type. Instead, the warning will relate to the fact that in the stopping test you are comparing a signed quantity (the loop index) to an unsigned one (the `size`). Doing so is considered an unsafe practice. For this and other reasons, the `ssize` function returns the size as a signed quantity.

### 10.9.3 Container copying

Using the `copy constructor` on containers such as vectors invokes the copy constructor of each individual element. Types such as `float` are called ‘trivially constructible’ or ‘trivially copyable’, and they are optimized for: copying a `vector<int>` is done by a `memcpy` or equivalent mechanism.

### 10.9.4 Failed allocation

If you ask for more data than your system can support, the allocation may fail, and a `bad_alloc` exception is thrown.

This is unlike the approach in C of returning `NULL` or `nullptr`.

### 10.9.5 Stack and heap allocation

Entities stored in memory (variables, structures, objects) can exist in two locations: the *stack* and the *heap*.

- Every time a program enters a new scope, entities declared there are placed on top of the stack, and they are removed by the end of the scope. Because of this automatic behavior, this is known as *automatic allocation*.
- By contrast, *dynamic allocation* creates a memory block that is not removed at the end of the scope, and so this block is placed on the heap. That block of memory can be returned to the free store at any time, so the heap can suffer from *fragmentation*.

#### 10.9.5.1 Illustrations in C

Automatic memory allocation, or the allocation of static memory, uses scopes, just like it does for the creation of scalars:

```
// assume there are no variables i,f,str here
{ // enter scope
    int i;
    float f;
    char str[5];
    // stuff
}
// the names i,f,str are unknown again here.
```

Objects that obey scope are allocated on the stack, so that their memory is automatically freed control leaves the scope. On the other hand, overuse of automatic allocation may lead to *stack overflow*.

Dynamic memory allocation was done by a call to `malloc`, and by assigning the returned memory address to a variable that was defined outside the scope, the block is known outside the scope:

```
double *array;
{ // enter scope
    array = malloc(5*sizeof(double));
    // exit scope
}
array[4] = 1.5; // this is legal
free(array); // release the malloc'ed memory
```

Dynamically created objects, such as the target of a pointer, live on the heap because their lifetime is not subject to scope.

The existence of the second category is a source of *memory leaks*, since it's too easy to forget the `free` call.

#### 10.9.5.2 Illustrations in C++

First of all, the `malloc` and `free` calls exist in C++, as do slightly more convenient variants `new/delete`:

```
double *array = new[5];
// stuff
delete array;
```

However, the idiomatic C++ way to create arrays with dynamically determined memory is by using `std::vector`.

```
int n = .... ;
{ // enter scope
    vector<int> array(n);
    // stuff
} // exit scope
```

This combines the best features of C allocation:

1. The storage for the vector is created on the heap, so you need not worry about stack overflow;
2. exiting the scope, both the definition of the vector goes away, and its dynamic memory is freed. This is technically known as *RAII*, and this mechanism is preferred as per C++ Core Guidelines [8], R.1.

If you absolutely have to manage dynamic allocation yourself, use `new/delete` instead of `malloc/free`, since the latter do not behave well with constructors; C++ Core Guidelines [8], R.10.

Dynamically allocated memory can transcend the scope it's created in:

```
vector<int> f(int n) {
    return vector<int>(n); // vector created inside function scope
};
vector<int> v;
v = f(5);
```

What happens here is the following:

1. A vector is created inside the function;
2. the `return` statement copies the vector, with all its data, to the variable `v` in the calling environment;
3. but by an optimization, the copy is omitted, and the actual memory is now assigned to the variable `v`.

In effect, we have now achieved a safer version of the function example of the above C section.

Another option for dynamic memory that is not scope-bound is to use the *smart pointer* mechanism, which also guarantees against memory leaks. See chapter 16.

### 10.9.6 Vector of bool

Booleans variables take a whole byte, even though a boolean strictly only needs a bit. However, you could optimize an array of bits, and thereby `vector<bool>`, by packing the bits into an integer, giving a factor of 8 savings in space.

Unfortunately, this optimization means that you can not get a reference to the elements.

```
vector<bool> bits;
for ( auto& b : bits ) // DOES NOT COMPILE
    b = false;
auto& f = bits.front(); // DOES NOT COMPILE
```

### 10.9.7 Span and mspan

#### 10.9.7.1 Installing span before C++20

```
clone the repo:
git clone https://github.com/martinmoene/gsl-lite.git

add to your compile line
-I${HOME}/Installation/gsl/gsl-lite/include (or whatever the path is to y

in your source:
#include "gsl/gsl-lite.hpp"
using gsl::span;
```

#### 10.9.7.2 mspan

In C++23 there is `mspan`, a multi-dimensional span, in the sense of it being a non-owning container-like object. Its main improvement over `span` is that it offers functionality for multi-dimensional indexing.

Header: `mspan`

Create 2D `mspan` from vector:

```
// mspan/index2std.cpp
// matrix in row major
vector<float> A(M*N);
std::mspan
```

```
Amd{ A.data(), std::extents{M,N} };
```

An `mdspan` object is a multi-dimensional Cartesian brick, where each dimension can be specified in a number of ways, the easiest being by giving its *extent* in that dimension

```
vector<float> ar10203040(10*20*30*40);
auto brick10203040 =
    std::mdspan< float, extents<10,20,30,40> >( ar10203040.data() );
auto mid = brick10203040[5,10,15,20];
```

Creation of `mdspan` objects is flexible. For instance, while the default memory ordering is *row major*, it is also possible to have *column major* ordering:

```
// mdspan/index2std.cpp
// matrix in column major
vector<float> B(M*N);
std::mdspan<float, std::extents<int, M,N>, std::layout_left>
    Bmd{B.data()};
```

**Remark** Introducing `mdspan` made it necessary to redefine the function of the comma inside square brackets. The comma operator, denoting sequential execution, can still be used in other contexts such as loop headers. If you absolutely need the sequential comma in an index expression, you can write `a[(i,j)]`.

The object now has an `extent` method to query the extents.

As an illustration of simple indexing, here is a rowsum calculation:

```
// mdspan/index2.cpp
int M = mat.extent(0); int N = mat.extent(1);
vector<float> rowsums(N);
for ( int row=0; auto& rs : rowsums ) {
    auto the_row =
        rng::iota_view(0,M)
        | rng::views::transform ( [mat,row] (int col) -> float {
            return mat[row,col]; } );
    rs = rng::accumulate( the_row, 0.f );
    row++;
}
```

In the type of operations that you are likely to do on multi-dimensional objects, you probably need multi-dimensional indices. This is provided by such methods as `ranges::cartesian_product`:

```
// mdspantranspose.cpp
// (i,j) index pairs into A and B
auto Aij = rng::views::cartesian_product
    ( rng::views::iota( 0L, static_cast<long int>(Amd.extent(0)) ),
```

```

rng::views::iota( 0L, static_cast<long int>(Amd.extent(1)) )
);

```

These indices can now be used for matrix-like indexing in the `mdspan` objects. For example, here is a matrix transposition, using `mdspan`, which is proposed for C++26, and here taken from the *Kokkos* library:

```

// mdspan/transpose.cpp
std::for_each
    ( std::execution::par_unseq,
      Aij.begin(), Aij.end(),
      [&] ( auto idx ) {
        auto [i,j] = idx;
        Bmd[j,i] = Amd[i,j];
      }
    );

```

#### 10.9.7.3 Installing `mdspan` from Kokkos

```

// mdspan/index2.cpp
#include "mdspan/mdspan.hpp"
namespace md = Kokkos;
namespace KokkosEx = MDSPAN_IMPL_STANDARD_NAMESPACE::MDSPAN_IMPL_PROPOSED;
namespace mdx = KokkosEx;

```

### 10.9.8 Size and signedness

The `size` method returns an unsigned quantity, so with a sufficiently high warning level

```
for (int i=0; i<myarray.size(); i++ )
```

will complain about mixing signed and unsigned quantities.

You can either

```
for (size_t i=0; i<myarray.size(); i++ )
```

or in C++20 use the `ssize` method, which returns a signed size:

```
for (int i=0; i<myarray ssize(); i++ )
```

<https://stackoverflow.com/questions/56217283/why-is-stdssize-introduced-in-c20#56217338>

## 10.10 C style arrays

Automatic arrays can be used as in C.

```
Code:  
// array/staticinit.cpp  
{  
    int numbers[] = {5, 4, 3, 2, 1};  
    println("{}, numbers[3]);  
}  
  
{  
    int numbers[5]{5, 4, 3, 2, 1};  
    numbers[3] = 21;  
    println("{}, numbers[3]);  
}
```

```
Output:  
[array] staticinit:  
2  
21
```

These has the (minimal) advantage of not having the overhead of a class mechanism. On the other hand, they have a number of disadvantages:

- You can not query the size of an array by its name: you have to store that information separately in a variable.
- Passing such an array to a function is really passing the address of its first element, so it is always (sort of) by reference.

### 10.10.1 Allocation

Traditionally, C arrays could only be allocated as

```
int a[5];  
float b[6][7];
```

that is, with explicitly given array bounds. Some compilers supported as an extension so-called *variable length arrays*:

```
int n; scanf("%d", &n); // this reads n from the console  
double x[n];
```

This mechanism was added to the C99 standard, but since support of it was not universal, the C11 standard made it optional again. The macro `__STC_NO_VLA__` is set to 1 if such support is indeed lacking.

Another thing to be aware of is that these arrays are allocated on the *stack*, so creating a too-large array may give stack overflow. This will make your code bomb with no informative error.

### 10.10.2 Indexing and range-based loops

Range-based indexing works the same as with vectors:

**Code:**

```
// array/rangemax.cpp
int numbers[] = {1, 4, 2, 6, 5};
int tmp_max = numbers[0];
for (auto v : numbers)
    if (v > tmp_max)
        tmp_max = v;
cout << "Max: " << tmp_max << " (should be
6)" << '\n';
```

**Output:**

[array] rangemax:  
Max: 6 (should be 6)

**Review 10.2.** The following codes are not correct in some sense. How will this manifest itself?

```
int a[5];
a[6] = 1.;

int a[5];
a.at(6) = 1.;
```

### 10.10.3 C-style arrays and subprograms

C-style arrays are really an abuse of the equivalence of arrays and addresses of the C programming language. This appears for instance in parameter passing mechanisms.

Arrays can be passed to a subprogram, but the bound is unknown there.

```
// pointer/arraypass.cpp
void set_array( double *x, int size) {
    for (int i=0; i<size; ++i)
        x[i] = 1.41;
}
/* ... */
double array[5] = {11, 22, 33, 44, 55};
set_array(array, 5);
cout << array[0] << "...." << array[4] << '\n';
```

**Exercise 10.19.** Rewrite the above exercises where the sorting tester or the maximum finder is in a subprogram.

Unlike with scalar arguments, array arguments can be altered by a subprogram: it is as if the array is always passed by reference. This is not strictly true: what happens is that the address of the first element of the array is passed. Thus we are really dealing with pass by value, but it is the array address that is passed rather than its value.

In subprograms, such arrays are indistinguishable from pointers. This is known as *pointer decay*. The following code and error message illustrates this:

**Code:**

```
// array/carray.cpp
void std_f( int stat[] ) {
    printf(.. in function: %lu\n",std::size(
        stat));
}
//codesnippet end
/* ... */
int stat[23];
std_f( stat );
```

**Output:**

```
[array] carray:
carray.cxx: In function
    ↪'void std_f(int*)':
carray.cxx:18:43: error: no
    ↪matching function for
    ↪call to 'size(int*&)'
18 |     printf(.. in
    ↪function:
    ↪%lu\n",std::size(stat));
    |
    ↪~~~~~^~~~~~
```

#### 10.10.4 Size of arrays

What does the `sizeof` operator give on various types of arrays?

**Code:**

```
// array/staticsize.cpp
int a1[10];
cout << "static: " << sizeof(a1) << '\n';
int *a2 = (int*) malloc( 10*sizeof(int) );
cout << "malloc: " << sizeof(a2) << '\n';
vector<int> a3(10);
cout << "vector: " << sizeof(a3) << '\n';
```

**Output:**

```
[array] staticsize:
static: 40
malloc: 8
vector: 24
```

You may think that `sizeof` on a C-style array is useful, but that doesn't survive passing to a subprogram:

**Code:**

```
// c/carray.c
void stat_f( int stat[] ) {
    printf(.. in function: %lu\n",sizeof(stat
        ));
}
//codesnippet
```

**Output:**

```
[c] carraystat:
carray.c:16:40: warning:
    ↪sizeof on array
    ↪function parameter will
    ↪return size of 'int *'
    ↪instead of 'int []'
    ↪[-Wsizeof-array-argument]
printf(.. in function:
    ↪%lu\n",sizeof(stat));

    ^
carray.c:15:18: note:
    ↪declared here
void stat_f( int stat[] ) {
    ^
1 warning generated.
Size of stat[23]: 92
.. in function: 8
```

Note the compiler warning

```
warning: sizeof on array function parameter will return size of 'int *' i
```

### 10.10.5 Multi-dimensional arrays

Multi-dimensional arrays can be declared and used with a simple extension of the prior syntax:

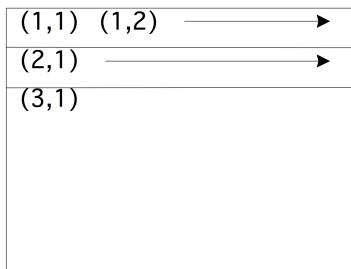
```
float matrix[15][25];

for (int i=0; i<15; i++)
    for (int j=0; j<25; j++)
        // something with matrix[i][j]
```

Passing a multi-dimensional array to a function, only the first dimension can be left unspecified:

```
// array/contig.cpp
void print12( int ar[][][6] ) {
    cout << "Array[1][2]: " << ar[1][2] << '\n';
    return;
}
/* ... */
int array[5][6];
array[1][2] = 3;
print12(array);
```

C/C++ row major



Physical:



### 10.10.6 Memory layout

Puzzling aspects of arrays, such as which dimensions need to be specified and which not in a function call, can be understood by considering how arrays are stored in memory. The question then is how a two-dimensional (or higher dimensional) array is mapped to memory, which is linear.

- A one-dimensional array is stored in contiguous memory.
- A two-dimensional array is also stored contiguously, with first the first row, then the second, et cetera.
- Higher dimensional arrays continue this notion, with contiguous blocks of the highest so many dimensions.

As a result of this, indexing beyond the end of a row, brings you to the start of the next row:

```
// array/contig.cpp
void print06( int ar[][][6] ) {
```

```

cout << "Array[0][6]: " << ar[0][6] << '\n';
return;
}
/* ... */
int array[5][6];
array[1][0] = 35;
print06(array);

```

We can now also understand how arrays are passed to functions:

- The only information passed to a function is the address of the first element of the array;
- In order to be able to find location of the second row (and third, et cetera), the subprogram needs to know the length of each row.
- In the higher dimensional case, the subprogram needs to know the size of all dimensions except for the first one.

## 10.11 Exercises

**Exercise 10.20.** Given a vector of integers, write two loops;

1. One that sums all even elements, and
2. one that sums all elements with even indices.

Use the right type of loop.

**Exercise 10.21.** Program *bubble sort*: go through the array comparing successive pairs of elements, and swapping them if the second is smaller than the first. After you have gone through the array, the largest element is in the last location. Go through the array again, swapping elements, which puts the second largest element in the one-before-last location. Et cetera.

*Pascal's triangle* contains binomial coefficients:

|     |     |                             |
|-----|-----|-----------------------------|
| Row | 1:  | 1                           |
| Row | 2:  | 1 1                         |
| Row | 3:  | 1 2 1                       |
| Row | 4:  | 1 3 3 1                     |
| Row | 5:  | 1 4 6 4 1                   |
| Row | 6:  | 1 5 10 10 5 1               |
| Row | 7:  | 1 6 15 20 15 6 1            |
| Row | 8:  | 1 7 21 35 35 21 7 1         |
| Row | 9:  | 1 8 28 56 70 56 28 8 1      |
| Row | 10: | 1 9 36 84 126 126 84 36 9 1 |

where

$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$

The coefficients can be computed from the recurrence

$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} & \end{cases}$$

(There are other formulas. Why are they less preferable?)

### Exercise 10.22.

- Write a class `pascal` so that `pascal(n)` is the object containing  $n$  rows of the above coefficients.  
Write a method `getvalue(i, j)` that returns the  $(i, j)$  coefficient.
- Write a method `print` that prints the above display.

The object needs to have an array internally. The easiest solution is to make an array of size  $n \times n$ . Optionally you can optimize your code to use precisely enough space for the coefficients.

The previous Pascal exercise did not separate the concerns of storage management and mathematics. In the following exercise we will handle them separately.

**Exercise 10.23.** Write a class `storage` that provides `get`/`set` methods that only read from and write to the data structure. The `pascal` class can then inherit from it, and do the coefficient calculation. Do you use public or private inheritance?

**Exercise 10.24.** Extend the `storage` class:

- If a coefficient outside the initial triangle is asked, the triangle should dynamically be extended to the row of that coefficient.
- This requires the `storage` class to extend the space for the coefficients.
- It also requires the `pascal` class to track how many rows have been filled in, and possibly compute some missing coefficients.

### Exercise 10.25.

- First print out the whole pascal triangle; then:
- Write a method `print(int m)` that prints a star if the coefficient modulo  $m$  is nonzero, and a space otherwise.

```

        *
      * *
    *   *
  * *   *
*   *       *
* *   *   *
*   *   *   *
* * *   *   *   *
*           *
*   *           *

```

- Accept any number of integers; for each, print out the triangle module that number. On zero: stop.

**Exercise 10.26.** A knight on the chess board moves by going two steps horizontally or vertically, and one step either way in the orthogonal direction. Given a starting position, find a sequence of moves that brings a knight back to its starting position. Are there starting positions for which such a cycle doesn't exist?

**Exercise 10.27.** From the ‘Keeping it REAL’ book, exercise 3.6 about Markov chains.

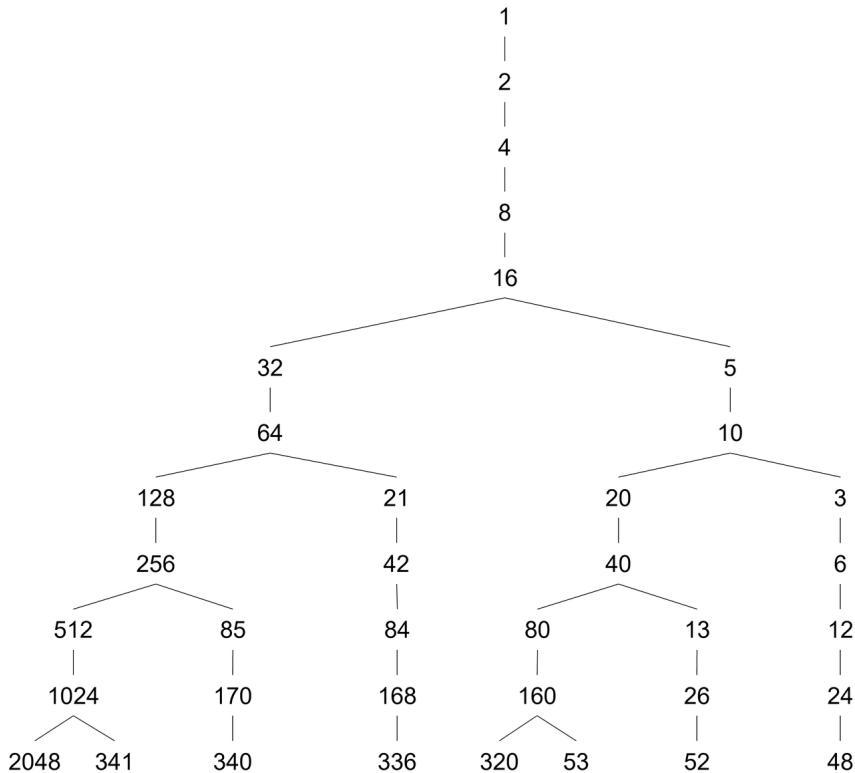


Figure 10.4: The ‘Collatz tree’

**Exercise 10.28.** Revisit exercise 6.14, and generate the ‘Collatz tree’ (figure 10.4): at level  $n$  (one-based counting) are the numbers that in  $n - 1$  steps converge to 1.

Read in a number  $n$  and print the first  $n$  rows, each row on a new line, with the numbers separated by spaces.

# Chapter 11

## Strings

### 11.1 Characters

- Type `char`;
- represents ‘7-bit ASCII’: printable and (some) unprintable characters.
- Single quotes: `char c = 'a'`

Equivalent to (short) integer:

Code:

```
// string/intchar.cpp
char ex = 'x';
int x_num = ex, y_num = ex+1;
char why = y_num;
cout << "x is at position " << x_num
    << '\n';
cout << "one further lies " << why
    << '\n';
```

Output:

```
[string] intchar:
x is at position 120
one further lies y
```

Also: `'x' - 'a'` is distance `a - x`

**Remark** The translation from `'x'` to ascii code, and in particular the letters having consecutive values, are not guaranteed by the standard.

**Exercise 11.1.** Write a program that accepts an integer  $1 \dots 26$  and prints the so-manieth letter of the alphabet.

Extend your program so that if the input is negative, it prints the minus-so-manieth uppercase letter of the alphabet.

### 11.2 Basic string stuff

## 11. Strings

---

```
#include <string>
using std::string;

// .. and now you can use 'string'

(Do not use the C legacy mechanisms.)
```

A *string* variable contains a string of characters.

```
string txt;
```

You can initialize the string variable or assign it dynamically:

```
string txt{"this is text"};
string moretxt("this is also text");
txt = "and now it is another text";
```

Normally, quotes indicate the start and end of a string. So what if you want a string with quotes in it?

You can escape a quote, or indicate that the whole string is to be taken literally:

**Code:**

```
// string/quote.cpp
string
one("a b c"),
two("a \"b\" c"),
three(R"(a ""b """c)" );
cout << one << '\n';
cout << two << '\n';
cout << three << '\n';
```

**Output:**

```
[string] quote:
a b c
a "b" c
"a ""b """c
```

Strings can be concatenated:

**Code:**

```
// string/strings.cpp
string my_string, space{" "};
my_string = "foo";
my_string += space + "bar";
cout << my_string << ":" << my_string.size()
<< '\n';
```

**Output:**

```
[string] stringadd:
foo bar: 7
```

You can query the *size*:

**Code:**

```
// string/strings.cpp
string five_text{"fiver"};
cout << five_text.size() << '\n';
```

**Output:**

```
[string] stringsize:
5
```

or use subscripts:

**Code:**

```
// string/stringsub.cpp
string digits{"0123456789"};
cout << "char three: "
    << digits[2] << '\n';
cout << "char four : "
    << digits.at(3) << '\n';
```

**Output:**  
**[string] stringsub:**  
*char three: 2  
char four : 3*

Same as ranging over vectors.

Range-based for:

**Code:**

```
// string/stringrange.cpp
cout << "By character: ";
for (char c : abc)
    cout << c << " ";
cout << '\n';
```

**Output:**  
**[string] stringrange:**  
*By character: a b c*

Ranging by index:

**Code:**

```
// string/stringrange.cpp
string abc = "abc";
cout << "By character: ";
for (int ic=0; ic<abc.size(); ic++)
    cout << abc[ic] << " ";
cout << '\n';
```

**Output:**  
**[string] stringindex:**  
*By character: a b c*

Range-based for makes a copy of the element

You can also get a reference:

**Code:**

```
// string/stringrange.cpp
for (char &c : abc)
    c += 1;
cout << "Shifted: " << abc << '\n';
```

**Output:**  
**[string] stringrangeset:**  
*Shifted: bcd*

```
for (auto c : some_string)
    // do something with the character 'c'
```

**Review 11.1.** True or false?

## 11. Strings

---

1. '0' is a valid value for a `char` variable single-quote 0 is a valid char+
2. "0" is a valid value for a `char` variable double-quote 0 is a valid char+
3. "0" is a valid value for a `string` variable double-quote 0 is a valid string+
4. 'a'+'b' is a valid value for a `char` variable adding single-quote chars is a valid char+

**Exercise 11.2.** The oldest method of writing secret messages is the *Caesar cipher*. You would take an integer  $s$  and rotate every character of the text over that many positions:

$$s \equiv 3: \text{"acdз"} \Rightarrow \text{"dfгc"}.$$

Write a program that accepts an integer and a string, and display the original string rotated over that many positions.

**Exercise 11.3.** (this continues exercise 11.2)

If you find a message encrypted with the Caesar cipher, can you decrypt it? Take your inspiration from the *Sherlock Holmes* story ‘The Adventure of the Dancing Men’, where he uses the fact that ‘e’ is the most common letter.

Can you implement a more general letter permutation cipher, and break it with the ‘dancing men’ approach?

Other methods for the vector class apply: `insert`, `empty`, `erase`, `push_back`, et cetera.

**Code :**

```
// string/strings.cpp
string five_chars;
cout << five_chars.size() << '\n';
for (int i=0; i<5; ++i)
    five_chars.push_back(' ');
cout << five_chars.size() << '\n';
```

**Output :**

```
[string] stringpush:
0
5
```

Methods only for `string`: `find` and such.

[http://en.cppreference.com/w/cpp/string/basic\\_string](http://en.cppreference.com/w/cpp/string/basic_string)

**Exercise 11.4.** Write a function to print out the digits of a number: 156 should print one five six. You need to convert a digit to a string first; can you think of more than one way to do that?

Start by writing a program that reads a single digit and prints its name.

For the full program it is easiest to generate the digits last-to-first. Then figure out how to print them reversed.

**Exercise 11.5.** Write a function to convert an integer to a string: the input 215 should give two hundred fifteen, et cetera.

**Exercise 11.6.** Write a pattern matcher, where a period . matches any one character, and `x*` matches any number of ‘x’ characters.

For example:

- The string abc matches a.c but abbc doesn't.
- The string abbc matches ab\*c, as does ac, but abzbc doesn't.

## 11.3 String streams

You can concatenate string with the + operator. The less-less operator also does a sort of concatenation. It is attractive because it does conversion from quantities to string. Sometimes you may want a combination of these facilities: conversion to string, with a string as result.

For this you can use a *string stream* from the `sstream` header.

Like cout (including conversion from quantity to string), but to object, not to screen.

- Use the `<<` operator to build it up; then
- use the `str` method to extract the string.

```
#include <sstream>
stringstream s;
s << "text" << 1.5;
cout << s.str() << endl;
```

## 11.4 Advanced topics

### 11.4.1 String views

Many times you will manipulate string objects without ever altering them. For that case, C++17 introduced `string_view` (in the `string_view` header) which gives you read-only views on a string. There are many operations on a string view, such as obtaining a new one by truncating so many characters at the front or back.

### 11.4.2 Raw string literals

You can include characters such as quotes or backslashes in a string by escaping them. This may get tiresome. The C++11 standard has a mechanism for *raw string literals*.

In its simplest form:

|                                                                                             |                                                        |
|---------------------------------------------------------------------------------------------|--------------------------------------------------------|
| <b>Code:</b>                                                                                | <b>Output:</b>                                         |
| <pre>// string/raw.cpp cout &lt;&lt; R"(string with ) { \weird\ stuf)" &lt;&lt; '\n';</pre> | <pre>[string] raw1: string with } { \weird\ stuf</pre> |

The obvious question is now of course how to include the closing-paren-quote sequence in a string. For this, you can specify your own multi character delimiter:

## 11. Strings

---

### Code:

```
// string/raw.cpp
    cout << R"limit("(string with )
{ \weird\ stuff)")limit" << '\n';
```

### Output:

```
[string] raw2:
"(string with )
{ \weird\ stuff)"
```

### 11.4.3 String literal suffix

A string literal "foo" is often compatible with a `std::string` but it is not of that type. Should you need that, you can add a suffix to the literal, which is defined in the namespace `string_literals`:

### Code:

```
// string/strings.cpp
void printfun( string s ) {
    cout << s << '\n';
}
void printfun_c( const string& s ) {
    cout << s << '\n';
}
/* ... */
using namespace std::string_literals;
printfun( "abc" );
printfun( "def"s );
printfun_c( "ghi" );
printfun_c( "jkl"s );
```

### Output:

```
[string] stringsuffix:
abc
def
ghi
jkl
```

### 11.4.4 Conversion to/from string

#### 11.4.4.1 Converting to string

There are various mechanisms for converting between strings and numbers.

- The C legacy mechanisms `sprintf` and `itoa`.
- `to_string`
- Above you saw the `stringstream`; section 11.3. Another use of this header follows below.
- The Boost library has a *lexical cast*.

Additionally, in C++17 there is the `charconv` header with `to_chars` and `from_chars`. These are low level routines that do not throw exceptions, do not allocate, and are each other's inverses. The low level nature is for instance apparent in the fact that they work on character buffers (not null-terminated). Thus, they can be used to build more sophisticated tools on top of.

#### 11.4.4.2 Converting from string

The `stringstream` object can be used to convert strings to numbers:

1. Initialize the string stream with a string;
2. Use a `cin`-like syntax to set a numerical variable from the stream.

```
// string/toint.cpp
string stringnum="12345";
int num;
stringstream numstream(stringnum);
numstream >> num;
```

### 11.4.5 Unicode

C++ strings are essentially vectors of characters. A character is a single byte. Unfortunately, in these interweb days there are more characters around than fit in one byte. In particular, there is the *Unicode* standard that covers millions of characters. The way they are rendered is by an *extendible encoding*, in particular *UTF8*. This means that sometimes a single ‘character’, or more correctly *glyph*, takes more than one byte.

## 11.5 C strings

In C a string is essentially an array of characters. C arrays don’t store their length, but strings do have functions that implicitly or explicitly rely on this knowledge, so they have a terminator character: ASCII **NULL**. C strings are called *null-terminated* for this reason.



## Chapter 12

### Input/output

Most programs take some form of input, and produce some form of output. In a beginning course such as this, the output is of more importance than the input, and all output is directed to the screen, so that's what we start by focusing on. We will also look at output to file, and input.

In examples so far, you have used `cout` with its default formatting. In this section we look at ways of customizing, and alternatives to, the `cout` output.

#### 12.1 Streams vs the format library

In the examples so far, you have mostly seen `cout` for screen output. This uses *streams*. Streams are very useful, but increasingly they should be considered a lower level mechanism.

In C++20 the `format` header uses a completely different syntax, close to Python's 'f-strings', and somewhat reminiscent of `printf` in C. Using this header has several advantages over streams, so we will discuss this as the preferred mechanism. However, since compilers are still catching up with the C++20 standard, we will also discuss stream formatting in detail.

**Remark** The `format` header is based on the open source `fmtlib` library. As compilers catch up with the C++20 and C++23 standards, you can use the functionality described below by using that library. In many cases, the only difference in your code lies in the header used:

Format header:

```
#include <format>
using std::format;
```

*Fmtlib:*

```
#include <fmt/format.h>
using fmt::format;
```

**Remark** If you only need to write a literal string (plus terminating newline) to standard out, the most efficient solution is with `std::puts`:

```
#include <cstdio>
...
std::puts("Hello world");
```

## 12.2 Using the `format` library

We start exploring the `format` library. The `std::format` command, introduced in C++20, gives a formatted string:

```
#include <format>

std::cout << std::format( /* .... */ );
```

but we will primarily use the ‘print’ functions from the C++23 `print` header:

```
#include <print> // C++23
```

The basic principle of formatting:

```
int i=2;
format("substituting {} brace expressions", i);
```

Specify format string, and arguments.

The basic structure of each `format` call is:

- The format string can have a number of placeholders indicated by braces;
- following it are various entities to be substituted for them.

The `format` routine returns a `string`, which you can write to screen using `std::cout`.

```
cout << format( /* ... */ );
```

In C++23 routines `std::print` and `std::println` have been added in the `print` header:

```
#include <print>
```

```
std::print( /* formatting stuff*/ ); // no newline unless you insert it
std::println( /* formatting stuff*/ ); // newline added
```

so that `cout` is no longer needed.

### 12.2.1 Numbered arguments

The basic `format` call uses empty braces as placeholders for values to the formatted. The braces can also contain modifiers. Here we show how a single number can indicate which argument to take; arguments can be inserted into more than one placeholder.

Argument mechanism:

- Arguments indicated by curly braces in the format string;
- braces can contain numbers (and modifiers, see next)

```
Code:
// iofmt/fmtbasic.cpp
println("{}", 2);
string hello_string = format
    ("{} {}!", "Hello", "world");
cout << hello_string << '\n';
println
    ("{}{}, {}{}!",
     "Hello", "world");
```

```
Output:
[iofmt] fmtbasic:
2
Hello world!
Hello, Hello world!
```

We now proceed to discuss various modifiers on the default formatting. For a listing see <https://en.cppreference.com/w/cpp/utility/format/spec>.

### 12.2.2 Align and padding

The ‘greater than’ sign plus a number indicates right aligning and the width of the field. Numbers that don’t fit that width ‘overflow’ on the right.

Right-align with > character and width:

```
Code:
// iofmt/fmtlib.cpp
for (int i=10; i<200000000; i*=10)
    fmt::print("{:>6}\n", i);
//cout << format("{:>6}\n",i);
```

```
Output:
[iofmt] fmtwidth:
      10
      100
      1000
      10000
      100000
      1000000
      10000000
      100000000
```

By default, the space character is used for padding a right-aligned string. A different padding character can be specified immediately before the greater-than sign.

Other than space for padding:

```
Code:
// iofmt/fmtlib.cpp
for (int i=10; i<200000000; i*=10)
    fmt::print("{:.>6}\n", i);
//cout << format("{:>6}\n",i);
```

```
Output:
[iofmt] fmtleftpad:
....10
...100
..1000
.10000
100000
1000000
10000000
100000000
```

### 12.2.3 Number bases

You can indicate the base with which to represent an integer by specifying one of `b`, `o`, `x` for binary, octal, hex respectively.

|                                                                                                                                                                              |                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <b>Code:</b><br><pre>// iofmt/fmtlib.cpp fmt::print     ("{} = {}{:b} bin\n", 17); fmt::print     ("{} = {}{:o} oct\n", 17); fmt::print     ("{} = {}{:x} hex\n", 17);</pre> | <b>Output:</b><br><b>[iofmt] fmtbase:</b><br><pre>17 = 10001 bin       = 21 oct       = 11 hex</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|

### 12.2.4 Floating point numbers

So far we have looked at integers. Floating point numbers, `float` and `double`, can be rendered a number of ways. First we consider ‘scientific notation’ which is forced with the `e` modifier, and a compact format that tries to avoid scientific notation if the number is not too large.

Floating point or normalized exponential with `e` specifier;  
Fixed: use decimal point if it fits, `m.n` specification

|                                                                                                                                                               |                                                                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Code:</b><br><pre>// iofmt/fmtfloat.cpp x = 1.234567; for (int i=0; i&lt;6; ++i) {     println         ("{}{:e}/{:e}\n",          x);     x *= 10; }</pre> | <b>Output:</b><br><b>[iofmt] fmtfloat:</b><br><pre>1.235e+00/ 1.235 1.235e+01/ 12.35 1.235e+02/ 123.5 1.235e+03/ 1235 1.235e+04/1.235e+04 1.235e+05/1.235e+05</pre> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|

If you’re printing out a column of numbers in scientific format, you will find that the above specifier makes the column wobble if there are both positive and negative number. An extra specifier indicates the treatment of any initial plus or minus:

- a plus sign will always print a leading plus or minus;
- a minus sign will print a minus sign for negative numbers, and nothing otherwise;
- a space will use a minus sign, or a space for nonnegative numbers.

Positive sign always, if needed, replace by blank:

```
Code:
// iofmt/fmtsci.cpp
float pi=3.14159f;
cout <<
    format("|{:+.2e}|{:+.2e}|\\n",
          pi,-pi);
cout <<
    format("|{:-.2e}|{:-.2e}|\\n",
          pi,-pi);
cout <<
    format("|{:.2e}|{:.2e}|\\n",
          pi,-pi);
```

```
Output:
[iofmt] fmtsci:
+3.14e+00	-3.14e+00
3.14e+00	-3.14e+00
3.14e+00	-3.14e+00
```

For comparison of numbers that are roughly of the same order of magnitude, it is convenient to list them in ‘fixed point’ notation, with the decimal point aligned. This is achieved by using the `f` modifier in the final location.

Use `f` modifier, give enough width, precision becomes number of decimal digits; compare `g` modifier for general:

```
Code:
// iofmt/fmtfix.cpp
float pi=3.14159f;
for ( auto x : { 0.543f, 1.13f, -21.55f, 33.4f
    , 156.8f } )
    println("{0:8g} : {0:-7.2f}",x);
```

```
Output:
[iofmt] fmtfix:
0.543 :      0.54
1.13  :     1.13
-21.55 : -21.55
33.4  :    33.40
156.8 : 156.80
```

## 12.2.5 Truth values

Booleans are by default printed as `true` or `false`:

```
format( "{}", true ); // gives 'true'
```

To get them printed as zero or one, use the `:d` modifier:

```
format( "{:d}", true ); // gives '1'
```

## 12.2.6 Limitations

The format library can not take any type to display. What it can display depends on what types the `std::formatter` is specialized for. Currently that’s mostly numerical types, character/string types, and types from the `chrono` library. Notable additions in C++23 are ranges.

However, many library types, notably `complex` and `bitset`, are still missing. You can write such specializations yourself; see section ??.

## 12.3 About fmtlib

The `format` library of the Standard Library comes out of the `fmtlib` library <https://github.com/fmtlib/fmt>. You may still use it for functions such as `print` that are not in every compiler.

The library can be discovered in CMake:

```
find_package( fmt )
target_link_libraries( program PUBLIC fmt::fmt )
```

### 12.3.1 Construct a string

If you want to construct a string piecemeal, for instance because it involves a loop over something, you can use a `memory_buffer`:

```
fmt::memory_buffer b;
fmt::format_to(std::back_inserter(b), "[");
for ( auto i : indices )
    fmt::format_to(std::back_inserter(b), "{}", ", i);
fmt::format_to(std::back_inserter(b), "]");
cout << to_string(b) << endl;
```

### 12.3.2 Output a range

In C++23, the fmtlib can immediately handle ranges:

```
auto rng = std::range::views::something;
std::print("{}\n", rng);
```

### 12.3.3 more

API documentation: <https://fmt.dev/latest/api.html>

Discoverability in CMake

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( FMTLIB REQUIRED fmt )
target_include_directories(
    program PUBLIC
    ${FMTLIB_INCLUDE_DIRS})

target_include_directories(
    prolib PUBLIC
    ${FMTLIB_INCLUDE_DIRS})
target_link_directories(
    prolib PUBLIC
    ${FMTLIB_LIBRARY_DIRS})
target_link_libraries(
    prolib PUBLIC fmt )
```

## 12.4 Stream-based formatting

### 12.4.1 Screen output

From `iostream`: `cout` uses default formatting.

Possible manipulation in `iomanip` header: pad a number, use limited precision, format as hex, etc.

Normally, output of numbers takes up precisely the space that it needs:

**Code:**

```
// io/io.cpp
for (int i=1; i<200000000; i*=10)
    cout << "Number: " << i << '\n';
```

**Output:**  
**[io] cunformat:**

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

We will now look at some examples of non-standard formatting. You may for instance want to output several lines, with the numbers in them nicely aligned. The most common I/O manipulation is to set a uniform width, that is, use the same number of positions for each number, regardless how many they need.

- The `setw` specifies how many positions to use for the following number.
- Note the singular in the previous sentence: the `setw` specifier applies only once.
- By default, numbers are right-aligned in the space given for them, and if they require more positions, they overflow on the right.

You can specify the number of positions, and the output is right aligned in that space by default:

**Code:**

```
// io/width.cpp
#include <iomanip>
using std::setw;
/* ... */
cout << "Width is 6:" << '\n';
for (int i=1; i<200000000; i*=10)
    cout << "Number: "
        << setw(6) << i << '\n';
cout << '\n';

// 'setw' applies only once:
cout << "Width is 6:" << '\n';
cout << ">"
    << setw(6) << 1 << 2 << 3 << '\n';
cout << '\n';
```

**Output:**  
**[io] width:**

```
Width is 6:
Number:      1
Number:     10
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000

Width is 6:
>      123
```

## 12. Input/output

---

Normally, padding is done with spaces, but you can specify other characters:

**Code:**

```
// io/formatpad.cpp
#include <iomanip>
using std::setfill;
using std::setw;
/* ... */
for (int i=1; i<200000000; i*=10)
    cout << "Number: "
        << setfill('.')
        << setw(6) << i
        << '\n';
```

**Output:**

```
[io] formatpad:
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Note: single quotes denote characters, double quotes denote strings.

Instead of right alignment you can do left:

**Code:**

```
// io/formatleft.cpp
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
/* ... */
for (int i=1; i<200000000; i*=10)
    cout << "Number: "
        << left << setfill('.')
        << setw(6) << i << '\n';
```

**Output:**

```
[io] formatleft:
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Finally, you can print in different number bases than 10:

**Code:**

```
// io/format16.cpp
#include <iomanip>
using std::setbase;
using std::setfill;
/* ... */
cout << setbase(16)
    << setfill(' ');
for (int i=0; i<16; ++i) {
    for (int j=0; j<16; ++j)
        cout << i*16+j << " ";
    cout << '\n';
}
```

**Output:**

```
[io] format16:
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

**Exercise 12.1.** Make the first line in the above output align better with the other lines:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
etc
```

Hex output is useful for addresses (chapter 17.2):

**Code:**

```
// pointer/coutpoint.cpp
int i;
cout << "address of i, decimal: "
    << (long)&i << '\n';
cout << "address of i, hex     : "
    << std::hex << &i << '\n';
```

**Output:**

```
[pointer] coutpoint:
address of i, decimal:
    ↪140732703427524
address of i, hex     :
    ↪0x7ffee2cbcfc4
```

Back to decimal:

## 12. Input/output

```
cout << hex << i << dec << j;
```

Binary output is not specified through a modifier. However, you can use the `bitset` container (from the `bitset` header) to print the bit pattern of an integer.

**Code:**

```
// io/bits.cpp
#include <bitset>
using std::bitset;
/* ... */
auto x255 = bitset<16>(255);
cout << x255 << '\n';
```

**Output:**

[io] **bits:**

```
0000000011111111
```

### 12.4.2 Floating point output

The output of floating point numbers is more tricky.

- How many positions are used for the digits before the decimal point?
- How many digits after the decimal point are printed?
- Is scientific notation used?

For floating point numbers, the `setprecision` modifier determines how many positions are used for the integral and fractional part together. If the integral part takes more positions, scientific notation is used.

Use `setprecision` to set the number of digits before and after decimal point:

**Code:**

```
// io/formatfloat.cpp
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
using std::setprecision;
/* ... */
x = 1.234567;
for (int i=0; i<10; ++i) {
    cout << setprecision(4) << x << '\n';
    x *= 10;
}
```

**Output:**

[io] **formatfloat:**

```
1.235
12.35
123.5
1235
1.235e+04
1.235e+05
1.235e+06
1.235e+07
1.235e+08
1.235e+09
```

This mode is a mix of fixed and floating point. See the `scientific` option below for consistent use of floating point format.

With the `fixed` modifier, `setprecision` applies to the fractional part.

Fixed precision applies to fractional part:

**Code:**

```
// io/fix.cpp
x = 1.234567;
cout << fixed;
for (int i=0; i<10; ++i) {
    cout << setprecision(4) << x << '\n';
    x *= 10;
}
```

**Output:**

[io] fix:  
1.2346  
12.3457  
123.4567  
1234.5670  
12345.6700  
123456.7000  
1234567.0000  
12345670.0000  
123456700.0000  
1234567000.0000

(Notice the rounding)

The `setw` modifier, for fixed point output, applies to the total width of integral and fractional part, plus the decimal point.

## Combine width and precision:

**Code:**

```
// io/align.cpp
x = 1.234567;
cout << fixed;
for (int i=0; i<10; ++i) {
    cout << setw(10) << setprecision(4) << x
        << '\n';
    x *= 10;
}
```

**Output:**

[io] align:  
1.2346  
12.3457  
123.4567  
1234.5670  
12345.6700  
123456.7000  
1234567.0000  
12345670.0000  
123456700.0000  
1234567000.0000

**Exercise 12.2.** Use integer output to print real numbers aligned on the decimal:**Code:**

```
// io/quasifix.cpp
string quasifix(double);
int main() {
    for (auto x : { 1.5, 12.32, 123.456,
                    1234.5678 } )
        cout << quasifix(x) << '\n';
```

**Output:**

[io] quasifix:  
1.5  
12.32  
123.456  
1234.5678

Use four spaces for both the integer and fractional part; test only with numbers that fit this format.

Above you saw that `setprecision` may give both fixed and floating point output. To get strictly floating point ‘scientific’ notation output, use `scientific`.

Combining width and precision:

**Code:**

```
// io/iof.cpp
x = 1.234567;
cout << scientific;
for (int i=0; i<10; ++i) {
    cout << setw(10) << setprecision(4)
        << x << '\n';
    x *= 10;
}
cout << '\n';
```

**Output:**

```
[io] iofsci:
1.2346e+00
1.2346e+01
1.2346e+02
1.2346e+03
1.2346e+04
1.2346e+05
1.2346e+06
1.2346e+07
1.2346e+08
1.2346e+09
```

### 12.4.3 Boolean output

The `boolalpha` modifier renders a `bool` variable as `true`, `false`.

### 12.4.4 Saving and restoring settings

```
ios::fmtflags old_settings = cout.flags();

cout.flags(old_settings);

int old_precision = cout.precision();

cout.precision(old_precision);
```

## 12.5 File output

The `iostream` is just one example of a `stream`, which is a general mechanism for converting entities to exportable form.

In particular, file output works the same as screen output: after you create a stream variable, you can ‘lessless’ to it.

```
mystream << "x: " << x << '\n';
```

The following example uses an `ofstream`: an output file stream. This has an `open` method to associate it with a file, and a corresponding `close` method.

### 12.5.1 Text output

Use:

**Code:**

```
// io/fio.cpp
#include <iostream>
using std::ofstream;
/* ... */
ofstream file_out;
file_out.open
    ("fio_example.out");
/* ... */
file_out << number << '\n';
file_out.close();
```

**Output:**

[io] fio:

```
echo 24 | ./fio ; \
          cat fio_example.out
A number please:
Written.
24
```

Compare: `cout` is a stream that has already been opened to your terminal ‘file’.

The `open` call can have flags, for instance for appending:

```
file.open(name, std::fstream::out | std::fstream::app);
```

### 12.5.2 Binary output

*binary I/O*

Binary output: write your data byte-by-byte from memory to file.  
(Why is that better than a printable representation?)

**Code:**

```
// io/fiobin.cpp
cout << "Writing: " << x << '\n';
ofstream file_out;
file_out.open
    ("fio_binary.out", ios::binary);
file_out.write
    (reinterpret_cast<char*>(&x),
     sizeof(double));
file_out.close();
```

**Output:**

[io] binout:

```
Writing: 0.841471
```

`write` takes an address and the number of bytes.

Input is mirror of the output:

|                                                                                                                                                                                                                                                                      |                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
| <b>Code:</b><br><pre>// io/fiobin.cpp ifstream file_in; file_in.open     ("fio_binary.out", ios::binary); file_in.read     (reinterpret_cast&lt;char*&gt;(&amp;x),      sizeof(double)); file_in.close(); cout &lt;&lt; "Read    : " &lt;&lt; x &lt;&lt; '\n';</pre> | <b>Output:</b><br><b>[io] binin:</b><br>Read    : 0.841471 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|

## 12.6 Output your own classes

The standard output mechanisms of C++ can only deal with primitive types and pointers. If you want to output objects, of your own devising or otherwise, it is possible to have them rendered, be it after a little coding.

### 12.6.1 Stream formatting

You have used statements like:

```
cout << "My value is: " << myvalue << "\n";
```

How does this work? The ‘double less’ is an operator with a left operand that is a stream, and a right operand for which output is defined; the result of this operator is again a stream. Recursively, this means you can chain any number of applications of << together.

If you want to output a class that you wrote yourself, you have to define how the << operator deals with your class.

Here we solve this in two steps:

|                                                                                                                                                                                                                                                                                                              |                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| Define a function that yields a string representing the object:<br><pre>#include &lt;iostream&gt; using std::stringstream;</pre><br><pre>// geom/pointfunc.cpp string as_string() {     stringstream ss;     ss &lt;&lt; "(" &lt;&lt; x &lt;&lt; ", " &lt;&lt; y &lt;&lt; ")";     return ss.str(); };</pre> | <pre>// geom/pointfunc.cpp string as_fmt_string() {     auto ss = format("({},{})", x, y);     return ss; };</pre> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|

Redefine the less-less operator:

```
// geom/pointfunc.cpp
std::ostream& operator<<
(std::ostream &out, Point &p) {
    out << p.as_string(); return out;
};
```

Use:

```
// geom/pointfunc.cpp
Point p1(1.,2.);
cout << "p1 " << p1
     << " has length "
     << p1.length() << "\n";
```

(See section 11.3 for `stringstream` and the `sstream` header.)

If you don't want to write that accessor function, you can declare the lessless operator as a `friend`:

```
class container {
private: double x;
public:
    friend ostream& operator<<( ostream& s, const container& c ) {
        s << c.x; /* no accessor */
        return s; };
};
```

## 12.7 Output buffering

In C, the way to get a newline in your output was to include the character `\n` in the output. This still works in C++, and at first it seems there is no difference with using `endl`. However, `endl` does more than breaking the output line: it performs a `std::flush`.

### 12.7.1 The need for flushing

Output is usually not immediately written to screen or disc or printer: it is saved up in buffers. This can be for efficiency, because output a single character may have a large overhead, or it may be because the device is busy doing something else, and you don't want your program to hang waiting for the device to free up.

However, a problem with buffering is the output on the screen may lag behind the actual state of the program. In particular, if your program crashes before it prints a certain message, does it mean that it crashed before it got to that line, or does it mean that the message is hanging in a buffer.

This sort of output, that absolutely needs to be handled when the statement is called, is often called *logging* output. The fact that `endl` does a flush would mean that it would be good for logging output. However, it also flushes when not strictly necessary. A better solution is to use `std::cerr` to send output to `stderr`; this works just like `cout`, except it doesn't buffer the output.

### 12.7.2 Performance considerations

If you want a newline in your output (whether screen or more general stream), using `endl` may slow down your program because of the flush it performs. More efficiently, you would add a newline character to the output directly:

```
somestream << "Value: " << x << '\n';
otherstream << "Total " << nerrors << " reported\n";
```

In other words, use `cout` for regular output, `cerr` for logging output, and use `\n` instead of `endl`.

## 12.8 Input

The `cin` command can be used to read integers and floating point formats.

**Code:**

```
// io/cinfloat.cpp
float input;
cin >> input;
cout << "(I think I got: " << input << ") \n"
;
```

**Output:**

```
[io] cinfloat:
for n in \
      1.5 1.6 1.67
→1.67e5 2.5.6 \
; do \
echo $n |
→./cinfloat \
; done
(I think I got: 1.5)
(I think I got: 1.6)
(I think I got: 1.67)
(I think I got: 167000)
(I think I got: 2.5)
```

As is illustrated with the last number in this example, `cin` will read until the first character that does not fit the format of the variable, in this case the second period. On the other hand, the `e` in the number before it is interpreted as the exponent of a floating point representation.

It is better to use `getline`. This returns a string, which you can parse later.

`getline` returns a string of the input line:

**Code:**

```
// io/gettermline.cpp
string input_string;
getline( cin, input_string );
printf
( "Input: <<{}>>", input_string );
```

**Output:**

```
[io] gettermline:
echo "one line, !!" \
| ./gettermline
Input: <<one line, !!>>
```

You can not use `cin` and `getline` in the same program.

`getline` returns a boolean, which you can use to test if you have reached the end of the stream:

**Code:**

```
// io/gettwoline.cpp
string input_string;
while (getline( cin, input_string ))
    println
        ( "Input: <<{}>>", input_string );
```

**Output:**

```
[io] gettwoline:
( echo "one line, !!" \
&& echo "2.line.2" ) \
| ./gettwoline
Input: <<one line, !!>>
Input: <<2.line.2>>
```

You can convert the string result of `getline` with the following bit:

```
// hello/helloinwhat.cpp
#include <iostream>
using std::cin;
using std::cout;
#include <sstream>
using std::stringstream;
/* ... */
std::string saymany;
int howmany;

cout << "How many times? ";
getline( cin, saymany );
stringstream saidmany(saymany);
saidmany >> howmany;
```

### 12.8.1 File input

Input file stream, method `open`, then use `getline` to read one line at a time:

```
// io/quickinput.cpp
#include <fstream>
using std::ifstream;
/* ... */
ifstream input_file;
input_file.open("fox.txt");
string oneline;
while (getline(input_file, oneline)) {
    cout << "Got line: <<" << oneline << ">>" << '\n';
}
```

There are several ways of testing for the end of a file

- For text files, the `getline` function returns `false` if no line can be read.
- The `eof` function can be used after you have done a read.
- `EOF` is a return code of some library functions; it is not true that a file ends with an EOT character. Likewise you can not assume a Control-D or Control-Z at the end of the file.

**Exercise 12.3.** Put the following text in a file:

```
the quick brown fox  
jummps over the  
lazy dog.
```

Open the file, read it in, and count how often each letter in the alphabet occurs in it

*Advanced note.* You may think that `getline` always returns a `bool`, but that's not true. It actually returns an `ifstream`. However, a conversion operator

```
explicit operator bool() const;
```

exists for anything that inherits from `basic_ios`.

### 12.8.2 Input streams

Tests, mostly for file streams: `is_eof` `is_open`

### 12.8.3 C-style file handling

The old `FILE` type should not be used anymore. The stream-based mechanisms in C++ have the big advantage that closing a file is done automatically at the end of a scope; C++ Core Guidelines [8], R.1.

### 12.8.4 Align and padding

In `fmtlib`, the ‘greater than’ sign plus a number indicates right aligning and the width of the field.

| Code:                                                                                                                                                     | Output:                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>// iofmt/fmtlib.cpp<br/>for (int i=10; i&lt;200000000; i*=10)<br/>    fmt::print("{:&gt;6}\n", i);<br/>//cout &lt;&lt; format("{:&gt;6}\n",i);</pre> | <pre>[io] fmtwidth:<br/>10<br/>100<br/>1000<br/>10000<br/>100000<br/>1000000<br/>10000000<br/>100000000<br/>1000000000<br/>1410065408<br/>1215752192</pre> |

**Code:**

```
// iofmt/fmtlib.cpp
for (int i=10; i<200000000; i*=10)
    fmt::print("{0.:>6}\n", i);
//cout << format("{0.:>6}\n",i);
```

**Output:**

[io] **fmtleftpad:**  
....10  
...100  
..1000  
.10000  
100000  
1000000  
10000000  
100000000

**12.8.5 Construct a string**

If you want to construct a string piecemeal, for instance because it involves a loop over something, you can use a *memory\_buffer*:

```
fmt::memory_buffer b;
fmt::format_to(std::back_inserter(b), "[");
for (auto i : indices)
    fmt::format_to(std::back_inserter(b), "{}", " ", i);
fmt::format_to(std::back_inserter(b), "]");
cout << to_string(b) << endl;
```

**12.8.6 Number bases**

In *fmtlib*, you can indicate the base with which to represent an integer by specifying one of `bin` for binary, `oct`, `hex` respectively.

**Code:**

```
// iofmt/fmtlib.cpp
fmt::print
    ("{} = {0:b} bin\n", 17);
fmt::print
    ("{} = {0:o} oct\n", 17);
fmt::print
    ("{} = {0:x} hex\n", 17);
```

**Output:**

[io] **fmtbase:**  
17 = 10001 bin  
= 21 oct  
= 11 hex

**12.8.7 Output your own classes**

With *fmtlib* this takes a different approach: here you need to specialize the `formatter` struct/class.

**Code:**

```
// iofmt/fmtlib.cpp
point p(1.1,2.2);
fmt::print("{}\n", p);
```

**Output:**

[io] **fmtstream:**  
(1.1,2.2)

### 12.8.8 Output a range

In C++23, the `fmtlib` can immediately handle ranges:

```
auto rng = std::range::views::something;
std::print("{}\n", rng);
```

## Chapter 13

### Lambda expressions

The mechanism of *lambda expressions* makes dynamic definition of functions possible.

Traditional function usage:

explicitly define a function and apply it:

```
float sum(float x, float y) { return x+y; }
cout << sum( 1.2f, 3.4f );
```

New:

apply the function recipe directly:

Code:

```
// lambda/lambdaex.cpp
[] (float x, float y) -> float {
    return x+y; } ( 1.5, 2.3 )
```

Output:

```
[lambda] lambdadirect:
3.8
```

This example illustrates the basic syntax of a lambda expression:

```
[capture] ( inputs ) -> outtype { definition };
```

- The square brackets, in this case, but not in general, empty, are the *capture* part;
- then follows the parenthesized parameter list;
- with a stylized arrow you can indicate the return type;
- and finally the usual function body, include `return` statement for non-void functions.

**Remark** *Lambda expressions are sometimes inaccurately called closures; this concept refers to the internal object containing the function recipe and captured environment.*

The above example can be classified as an *immediately invoked lambda* expression. While that example was somewhat pointless, there are uses for this idiom, such as being able to choose dynamically between two constructors.

Invoke different constructors based on runtime condition:

## 13. Lambda expressions

Does not work:

```
// lambda/invite.cpp
if (foo)
    MyClass x(5,5);
else
    MyClass x("foo");
```

Solution:

```
// lambda/invite.cpp
auto x =
    [foo] () {
        if (foo)
            return MyClass(5,5);
        else
            return MyClass("foo");
    }();
```

Note the use of **auto** and the omitted return type.

For a slightly more useful example, we can assign the lambda expression to a variable, and repeatedly apply it.

Lambda expression assigned to a variable:

Code:

```
// lambda/lambdaex.cpp
auto summing =
    [] (float x, float y) -> float {
        return x+y;
    };
cout << summing ( 1.5, 2.3 ) << '\n';
cout << summing ( 3.7, 5.2 ) << '\n';
```

Output:

```
[lambda] lambdavar:
3.8
8.9
```

- This is a variable declaration.
- Uses **auto** for technical reasons; see later.

Return type could have been omitted:

```
auto summing =
    [] (float x, float y) { return x+y; };
```

You can now exercise this by writing a toy numerical library.

**Exercise 13.1.** Do exercise 47.10 of the zero finding

### 13.1 Lambda expressions as function argument

Above, when we assigned a lambda expression to a variable, we used **auto** for the type. The reason for this is that each lambda expression gets its own unique type that is dynamically generated. Now we have a problem if we want to pass that variable to a function.

Suppose we want to pass a lambda expression to a function:

```
int main() {
    somefun( [] (int i) { cout << i+1; } );
```

What type do we use for the function parameter?

```
void somefun( /* what type are we giving? */ f ) {
    f(5);
}
```

Since the type of the lambda expression is dynamically generated, we can not specify that type in the function declaration.

The way out is to use the `functional` header:

```
#include <functional>
using std::function;
```

With this, you can declare parameters by their signature.

In the following example we write a function `apply_to_5` which

- takes a function `f`, and
- applies it to 5.

We call the `apply_to_5` function with a lambda expression as argument:

|                                                                                                                                                                       |                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| <b>Code:</b>                                                                                                                                                          | <b>Output:</b>                         |
| <pre>// lambda/lambdaex.cpp void apply_to_5 ( function&lt; void(int) &gt; f ) {     f(5); } /* ... */ apply_to_5 ( [] (int i) {     println("Int: {}", i); } );</pre> | <pre>[lambda] lambdapass: Int: 5</pre> |

Another possibility for passing function is to templatize

|                                                             |
|-------------------------------------------------------------|
| <b>Exercise 13.2.</b> Do exercise 47.11 of the zero-finding |
|-------------------------------------------------------------|

### 13.1.1 Lambda members of classes

The fact that a lambda expression has a dynamically generated type also makes it hard to store it in an object. To do this we again use `std::function`.

In the following example we make a class `SelectedInts` which takes a boolean function in the constructor: an object will contain only those integers that satisfy the function.

|                                                               |
|---------------------------------------------------------------|
| A set of integers, with a test on which ones can be admitted: |
|---------------------------------------------------------------|

## 13. Lambda expressions

Class:

```
// lambda/lambdafun.cpp
#include <functional>
using std::function;
/* ... */
class SelectedInts {
private:
    vector<int> bag;
    function< bool(int) > selector;
public:
    SelectedInts
        ( function< bool(int) > f ) {
            selector = f; }
    }
```

Methods:

```
// lambda/lambdafun.cpp
void add(int i) {
    if (selector(i))
        bag.push_back(i);
}
int size() {
    return bag.size(); }
std::string as_string() {
    std::string s;
    for ( int i : bag )
        s += to_string(i)+" ";
    return s;
}
```

We use the above class to construct an object as follows:

- we read an integer *divisor*,
- and accept only those integers into our object that are divisible by that number.

For this we write a lambda expression *is\_divisible* that

- captures the divisor, and then
- takes an integer as (its only) argument,
- returning whether that argument is divisible.

The above code in use:

Code:

```
// lambda/lambdafun.cpp
cout << "Give a divisor: ";
cin >> divisor;
cout << "... divisor {" << divisor;
auto is_divisible =
    [divisor] (int i) -> bool {
        return i%divisor==0; };
SelectedInts multiples( is_divisible );
for (int i=1; i<50; ++i)
    multiples.add(i);
```

Output:

```
[lambda] lambdafun:
Give a divisor:
.. using 7
Multiples of 7:
7 14 21 28 35 42 49
```

### 13.2 Captures

A *capture* is a way to ‘bake variables into’ a function. Let’s say we want a function that increments its input, and the increment amount is set when we define the function.

Increment function:

- scalar in, scalar out;
- the increment amount has been fixed through the capture.

```
Code:
// lambda/lambdacapture.cpp
int n;
cin >> n;
auto increment_by_n =
[n] ( int input ) -> int {
    return input+n;
};
cout << increment_by_n (5);
cout << increment_by_n (12);
cout << increment_by_n (25);
```

```
Output:
[lambda] lambdavalue:
(input value: 1)
6
13
26
```

The captured value is copied into the lambda expression. If you capture a variable, and change it afterwards, the lambda expression does not change. The reason is that the captured value is copied; below you will see that there is a possibility of capturing by reference.

Illustrating that the capture variable is copied once and for all:

```
Code:
// lambda/lambdaconstant.cpp
int inc;
cin >> inc;
auto increment =
[inc] ( int input ) -> int {
    return input+inc;
};
cout << "increment by: {" << inc;
cout << "1 -> {" << increment(1);
inc = 2*inc;
cout << "1 -> {" << increment(1);
```

```
Output:
[lambda] lambdaconstant:
increment by: 2
1 -> 3
1 -> 3
```

**Exercise 13.3.** Write a program that

- reads a `float` factor;
- defines a function `multiply` of one argument that multiplies its input by that factor.

You can capture more than one variable. Explicitly capturing variables is done with a comma-separated list.

Example: multiply by a fraction.

```
int d=2, n=3;
times_fraction = [d,n] (int i) ->int {
    return (i*d)/n;
}
```

**Exercise 13.4.**

- Set two variables

```
float low = .5, high = 1.5;
```

- Define a function `is_in_range` of one variable that tests whether that variable is between `low, high`.  
(Hint: what is the signature of that function? What is/are input parameter(s) and what is the return result?)

Use this again in the numerical library you are developing.

**Exercise 13.5.** Do exercises 47.12 and 47.13 of the zero-finding

### 13.2.1 Capture by reference

Normally, captured variables are copied by value.

Attempting to change the captured variable doesn't even compile:

```
auto f = // WRONG DOES NOT COMPILE
[x] ( float &y ) -> void {
    x *= 2; y += x; };
```

If you do want to alter the captured parameter, pass it by reference:

**Code:**

```
// lambda/lambdacapture.cpp
int stride = 1;
auto more_and_more =
    [&stride] ( int input ) -> void {
        cout << input << " => " << input+stride;
        ++stride;
    };
more_and_more(5);
more_and_more(6);
more_and_more(7);
more_and_more(8);
more_and_more(9);
cout << "stride is now: " << stride;
```

**Output:**

```
[lambda] lambdacapture:
(input value: 1)

5 => 6
6 => 8
7 => 10
8 => 12
9 => 14
stride is now: 6
```

Capturing by reference can for instance be useful if you are performing some sort of reduction. The capture is then the reduction variable, and the numbers to be reduced come in as function parameter to the lambda expression.

In this example we count how many of the input values test true under a certain function `f`:

Capture a variable by reference so that you can update it:

```
Code:
// lambda/countif.cpp
int count=0;
auto count_if_f =
[&count] (int i) {
    if (i%2==0) count++;
}
for (int i : {1,2,3,4,5} )
    count_if_f(i);
cout << "We counted: {" << count;
```

**Output:**  
**[lambda] countif:**  
*We counted: 2*

The last example used the `algorithm` header; section 14.3.

### 13.2.2 Capturing ‘this’

In addition to capturing specific variable, whether by reference or not, as you saw above, you can also capture the whole environment of a lambda. For this the following shorthands exist:

```
[=] () {} // capture everything by value
[&] () {} // capture everything by reference
```

As of C++20, implicit capture of `this` by value is deprecated: writing

```
[=] (params) { /* ... */ }
```

would capture `this` by value, but noting that `this` is a pointer, its members are actually captured by reference. Therefore, either of the follow should be written

```
[=,*this] (params) { /* ... */ };
[=&this] (params) { /* ... */ };
```

which capture members by value and reference respectively.

## 13.3 More

### 13.3.1 Making lambda stateful

Let’s consider the issue of lambda expressions and mutable state, by which we mean no more than that a variable internal to the lambda expression can get updated. Above you have already seen how you can capture a variable from the calling environment by reference.

Lambda expressions are normally stateless, meaning that the capture is captured by value, and is in fact `const`:

```
Code:
// lambda/mutable.cpp
float x = 2, y = 3;
auto f = [x] ( float &y ) -> void {
    int xx = x*2; y += xx; };
f(y);
cout << y << '\n';
f(y);
cout << y << '\n';
```

**Output:**  
**[lambda] nonmutable:**  
*7  
11*

## 13. Lambda expressions

---

You can make the capture non-`const`, and thereby make the lambda expression stateful, with the `mutable` keyword:

**Code :**

```
// lambda/mutable.cpp
float x = 2, y = 3;
auto f = [x] ( float &y ) mutable -> void {
    x *= 2; y += x; };
f(y);
cout << y << '\n';
f(y);
cout << y << '\n';
```

**Output :**  
[lambda] yesmutable:  
7  
15

How about if that count is not really needed in the calling environment of the lambda expression; can we somehow make it internal? Indeed, we can initialize local variables with a *capture-default* which in this case doesn't really capture anything.

```
[count=0] ( /* arguments */ ) mutable {
    count++; /* other statements */
}
```

Here is a nifty application: printing a list of numbers, separated by commas, but without trailing comma:

**Code :**

```
// lambda/lambdaexch.cpp
vector x{1,2,3,4,5};
auto printdigit =
    [start=true] (auto xx) mutable -> string{
        if (start) {
            start = false;
            return to_string(xx);
        } else
            return ","+to_string(xx);
    };
for ( auto xx : x )
    cout << printdigit(xx);
cout << '\n';
```

**Output :**  
[lambda] lambdaexch:  
1,2,3,4,5

### 13.3.2 Generic lambdas

The `auto` keyword can be used for *generic lambdas*:

```
auto compare = [] (auto a,auto b) { return a<b; };
```

Here the return type is clear, but the input types are generic. This is much like using a templated function: the compiler instantiates the expression with whatever types are needed from the context.

### 13.3.3 Algorithms

The `algorithm` header contains a number of functions that naturally use lambdas. For instance, `any_of` can test whether any element of a `vector` satisfies a condition. You can then use a lambda to specify the `bool` function that tests the condition.

This uses mechanisms we haven't discussed yet, so we postpone this until section 14.3.

### 13.3.4 C-style function pointers

The C language had a – somewhat confusing – notation for function pointers. If you need to interface with code that uses them, it is possible to use lambda functions to an extent: lambdas without captures can be converted to a function pointer.

Lambda expression with empty capture are compatible with C-style function pointers:

**Code:**

```
// lambda/lambda.cpp
int cfun_add1( int i ) {
    return i+1; }
int apply_to_5( int (*f) (int) ) {
    return f(5); }
/* ... */
auto lambda_add1 =
    [] (int i) { return i+1; };
cout << "C ptr: "
    << apply_to_5(&cfun_add1)
    << '\n';
cout << "Lambda: "
    << apply_to_5(lambda_add1)
    << '\n';
```

**Output:**

```
[lambda] lambda.cpp:
C ptr: 6
Lambda: 6
```



## Chapter 14

### Iterators, Algorithms, Ranges

You have seen how you can iterate over a vector

- by an indexed loop over the indices, and
- with a range-based loop over the values.

There is a third way, which is actually the basic mechanism underlying the range-based looping. For this you need to realize that iterating through objects such as vectors isn't simply a process of keeping a counter that says where you are, and taking that element if needed.

Many C++ classes have an *iterator* subclass, that gives a formal description of ‘where you are’ and what you can find there. Having iterators means that you can traverse structures that don’t have an explicit index count such as sets or maps, but there are many other conveniences as well.

Here are some ways iterator are similar to indexes:

- Iteratable containers have a `begin` and `end` iterator.
- The end iterator ‘points’ just beyond the last element.
- The ‘`*`’ star operator gives the element that the iterator points to.
- You can increment and decrement them (for certain containers).

We start with a discussion of ranges that does not involve iterators, and then we go on to the more general and basic mechanism.

#### 14.1 Ranges

The C++20 standard contains a `ranges` header, which generalizes iterable objects into as-it-were streams. This allows you to express algorithms in terms of operations on whole ranges, rather than on individual elements.

We need to introduce two new concepts: ranges and views. First of all, a range is something you can iterate over. The containers of the pre-17 standard library such as `std::vector` are ranges and you can iterate over them in range-based loops, but in this chapter you’ll learn other ways of ranging.

The `ranges` header contains a number of algorithms. Here are two examples, sorting and summing a vector. In both cases no iteration is explicitly mentioned:

Sorting:

```
// rangestd/sort.cpp
vector<int> v{3,1,2};
rng::sort(v);
for ( auto e : v )
    cout << e << '\n';
```

Summing:

```
// range/sumsquare.cpp
vector<float> elements{.5f,1.f,1.5f};
auto sum_of_elts =
    rng::accumulate( elements, 0.f );
cout << "Sum of elements: "
    << sum_of_elts << '\n';
```

In the examples to follow we will abbreviate

```
#include <ranges>
namespace rng = std::ranges;
```

so that you can write

```
rng::transform
```

and such.

However, some features are taken from the *range-v3* library (section ??), in which case

```
#include <range/v3/all.hpp>
namespace rng = ranges;
```

#### 14.1.1 Introduction

Let's consider some simple examples in which we combine: a rangable container, a lambda expression to apply to each element, and an algorithm that combines these. At first we use `for_each` to output each element of a vector.

With

```
// rangestd/range.cpp
vector<int> v{2,3,4,5,6,7};
```

**Code:**

```
// rangestd/range.cpp
#include <ranges>
namespace rng = std::ranges;
#include <algorithm>
/* ... */
vector<int> v{2,3,4,5,6,7};
rng::for_each
( v, [] (int i) { cout << i << " "; } );
```

**Output:**

```
[rangestd] cout:
2 3 4 5 6 7
```

Often we use a *capture by reference*, for instance to let a variable act as an accumulator. In the following example, the lambda expression is applied to each data element, and updates a global variable that was captured by reference.

Capture a global accumulator by reference:

**Code:**

```
// rangestd/range.cpp
count = 0;
rng::for_each
( v,
  [&count] (int i) {
    count += (i<5);
  });
cout << "Under five: "
<< count << '\n';
```

**Output:**  
**[rangestd]** count:  
*Under five: 3*

Let's use ranges to reformulate some loop-based code.

**Exercise 14.1.** Revisit the vector normalization and rewrite the `norm` function to use a `for_each` algorithm.

Also rewrite the `scale` function using a `for_each` algorithm.

### 14.1.2 Views

A view is somewhat similar to a range, in the sense that you can iterate over it. The difference is that, unlike for instance a `vector`, a view is not a completely formed object: its elements are formed as needed, often by applying a *range adapter* to a range. You would typically write something like:

`yourcontainer | someview`

and the result of this is just as iterable as the original container. The ‘view’ is often a *range adaptor*, taken from the `views` namespace. For instance:

`myvector | range::views::drop(5)`

which acts as if you omitted the first 5 elements from your vector.

Views act like ranges, for instance in the sense that you can iterate over them.

Filter a range by some condition:

```
Code:
// rangestd/filter.cpp
vector<float> numbers
{1,-2.2,3.3,-5,7.7,-10};
for ( auto n :
    numbers
    | std::ranges::views::filter
        ( [] (float f) -> bool {
            return f>0; } )
    )
cout << n << " ";
cout << '\n';
```

```
Output:
[rangestd] filter:
1 3.3 7.7
```

A view doesn't own any data, and any elements you view in it get formed as you iterate over it. This is sometimes called *lazy evaluation* or *lazy execution*. Stated differently, its elements are constructed as they are being requested by the iteration over the view. This even makes it possible to have infinite ranges, for example with `iota_view` (see section 14.1.4): everything works as long as you don't ask for the last element.

Views are composable: you can take one view, and pipe it into another one. Because of the lazy evaluation, no temporary objects corresponding to the intermediate ranges and views are formed. If you need the resulting object, rather than the elements as a stream, you can call (in C++23) `ranges::to`:

```
auto newvector = myvector | views::drop(5) | to<vector<int>>();
```

Let's look at some examples of views.

**Exercise 14.2.** Change the filter example to let the lambda count how many elements were  $> 0$ .

We start with a `transform` view, which applies a function to each element of the range or view in sequence. In this particular example we multiply each element by two.

```
Code:
// range/filtertransform.cpp
vector<int> v{ 1,2,3,4,5,6 };
cout << "Original data:\n "
<< vector_as_string(v)
<< '\n';
auto times_two = v
| rng::views::transform
( [] (int i) {
    return 2*i; } );
cout << "Times two:\n ";
for ( auto c : times_two )
    cout << c << " "; cout << '\n';
```

```
Output:
[range] ft0:
Original vector:
1, 2, 3, 4, 5, 6,
Times two:
2 4 6 8 10 12
```

Next we use `filter` on the transform results, which only yields those elements that satisfy some boolean test.

```
Code:
// range/filtertransform.cpp
auto over_five = times_two
| rng::views::filter
( [] (int i) {
    return i>5; } );
cout << "Over five: ";
for ( auto c : over_five )
    cout << c << " "; cout << '\n';
```

```
Output:
[range] ft1:
Over five: 6 8 10 12
```

In the above examples `times_two` and `over_five` are views, and explicitly formed and stored objects. The required elements in the loops are generated from the initial range object as the iteration requires. In particular, no temporary object of size comparable to the initial range are created.

We can collapse the two above snippets into one by composition of views:

```
Code:
// range/filtertransform.cpp
vector<int> v{ 1,2,3,4,5,6 };
/* ... */
auto times_two_over_five = v
| rng::views::transform
( [] (int i) {
    return 2*i; } )
| rng::views::filter
( [] (int i) {
    return i>5; } );
```

```
Output:
[range] ft2:
Original vector:
1, 2, 3, 4, 5, 6,
Times two over five:
6 8 10 12
```

Let's exercise this piping of containers and views.

**Exercise 14.3.** Make a vector that contains both positive and negative numbers. Use ranges to compute the minimum square root of the positive numbers.

1. Start with a vector of numbers;
2. Make an iterable view containing just the square roots of the positive numbers;
3. Find the minimum of these roots.

Other available operations:

- dropping initial elements: `std::views::drop`
- reversing a vector: `std::views::reverse`

In C++23 some more views have been added. For instance, if you have to iterate simultaneously over two containers, you can use the `ranges::zip` view. Here each iteration gives a tuple of elements from the zipped containers.

Zip ranges together with `ranges::views::zip`, giving a tuple:

```
Code:
// rangestd/zip.cpp
vector a { 10, 20, 30, 40, 50 };
vector<string> b
{ "one", "two", "three", "four" };

for (const auto& [num, name] :
    rng::views::zip(a, b))
cout << name << " -> " << num << '\n';
```

Output:  
[range] zip:  
one -> 10  
two -> 20  
three -> 30  
four -> 40

### Remark zip\_with zip\_transform

Return adjacent elements with `ranges::adjacent` as a tuple.

```
// range/adjacent.cpp
for (auto [ lt,mi,rt ] : values | views::adjacent<3> )
    cout << fixed
        << setw(3) << lt << ","
        << setw(3) << mi << ","
        << setw(3) << rt << '\n';
```

### 14.1.3 Example: sum of squares

For computing the sum of squares of the elements of a vector, we can use the range `transform` method for constructing a ‘lazy container’ of the squares. We use the `accumulate` adapter from C++23.

```
// range/sumsquare.cpp
vector<float> elements{.5f,1.f,1.5f};
auto squares =
    rng::views::transform(elements, [] (auto e) { return e*e; } );
auto sumsq =
    rng::accumulate(squares, 0.f);
cout << "Sum of squares: " << sumsq << '\n';
```

### 14.1.4 Infinite sequences

Since views are lazily constructed, it is possible to have an infinite object – as long as you don’t ask for its last element.

In the following example we make a view `even_numbers` that ‘contains’ all even numbers, and then we print only the first ten of them:

**Code:**

```
// range/infinite.cpp
auto even_numbers =
    rng::views::iota(0)
    | rng::views::filter
        ([] ( auto n ) { return n%2==0; } );
for ( auto n : even_numbers
    | rng::views::take(10) )
    cout << n << '\n';
```

**Output:**

```
[range] infinite:
0
2
4
6
8
10
12
14
16
18
```

Here we used the range version of `iota` in the variant where only the lower bound is specified, and we use `ranges::views::take` to take only the head of this list.

## 14.2 Iterators

Many algorithms that you saw above have an older syntax using iterators.

```
vector data{2,3,1};

// range syntax
ranges::sort(data);

// iterator syntax:
sort( begin(data), end(data) );
```

The `begin` / `end` functions give *iterators*, which are objects

### 14.2.1 Iterating with iterators

The container class has a subclass *iterator* that can be used to iterate through all elements of a container.

An `iterator` can be used outside of strictly iterating. You can consider an iterator as a sort of ‘pointer into a container’, and you can move it about.

Let’s look at some examples of using the `begin` and `end` iterators. In the following example:

- We first assign the `begin` and `end` iterators to variables; the `begin` iterator points at the first element, but the `end` iterator points just beyond the last element;
- Given an iterator, you get the value of the corresponding element by applying the ‘star’ operator to it;
- A sort of ‘pointer arithmetic’ can be applied to iterators.

Use independent of looping:

```
Code:
// stl/iter.cpp
vector<int> v{1,3,5,7};
auto pointer = v.begin();
cout << "we start at "
    << *pointer << '\n';
++pointer;
cout << "after increment: "
    << *pointer << '\n';

pointer = v.end();
cout << "end is not a valid element: "
    << *pointer << '\n';
pointer--;
cout << "last element: "
    << *pointer << '\n';
```

```
Output:
[stl] iter:
we start at 1
after increment: 3
end is not a valid element: 0
last element: 7
```

Note that the star notation is a *unary star operator* on the iterator object, not a *pointer dereference*:

```
// iter/iter.cpp
vector<int> vec{11,22,33,44,55,66};
auto second = vec.begin(); ++second;
cout << "Dereference second: "
    << *second << '\n';
// DOES NOT COMPILE
// the iterator is not a type-star:
// int *subarray = second;
```

Another illustration of iterators is getting the last element of a vector: We start by getting the value through the **back** method.

Use **back** to get the value of the last element:

```
Code:
// array/vectorend.cpp
vector<int> mydata(5,2);
// last element:
cout << mydata.back()
    << '\n';
mydata.push_back(35);
cout << mydata.size()
    << '\n';
// last element:
cout << mydata.back()
    << '\n';
```

```
Output:
[array] vectorend:
6
35
```

Next we use the iterator mechanism. The **end** iterator points just ‘beyond’ the data, so we shift it left and get its value.

Set an iterator to the last element and ‘dereference’ it:

|                                                                                                                                                                                      |                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| <b>Code:</b>                                                                                                                                                                         | <b>Output:</b>                             |
| <pre>// array/vectorend.cpp vector&lt;int&gt; mydata(5, 2); mydata.push_back(35); cout &lt;&lt; mydata.size() &lt;&lt; '\n'; cout &lt;&lt; *(*(--mydata.end())) &lt;&lt; '\n';</pre> | <pre>[array] vectorenditerator: 6 35</pre> |

### 14.2.2 Why still iterators, why not

One reason that range-based algorithms are better than iterator-based ones, is that the iterator version is prone to accidents:

```
sort( begin(data1), end(data2) ); // OUCH
```

There are still some things that you can do with iterators that can not be done with range-based iteration.

Reverse iteration can not be done with range-based syntax.

Use general syntax with reverse iterator: `rbegin`, `rend`.

### 14.2.3 Vector operations through iterators

You have already seen that the length of a vector can be extended with the `push_back` method by a single element.

With iterators other operations are possible, such as copying, erasing, and inserting.

First we show the use of `copy` which takes two iterators in one container to define the range to be copied, and one iterator in the target container, which can be the same as the source. The copy operation will overwrite elements in the target, but without bound checking, so make sure there is enough space.

Copy a begin/end range of one container to an iterator in another container::

|                                                                                                                                                                                                                                                                            |                                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|
| <b>Code:</b>                                                                                                                                                                                                                                                               | <b>Output:</b>                  |
| <pre>// iter/iter.cpp vector&lt;int&gt; counts{1,2,3,4}; vector&lt;int&gt; copied(5); copy( counts.begin(), counts.end(),       copied.begin() ); cout &lt;&lt; copied[0]      &lt;&lt; ", " &lt;&lt; copied[1]      &lt;&lt; ".." &lt;&lt; copied[4] &lt;&lt; '\n';</pre> | <pre>[iter] copy: 0, 1..4</pre> |

(No bound checking, so be careful!)

The erase operation `erase` takes two iterators, defining the inclusive lower and exclusive upper bound for the range to erase.

Erase from start to before-end:

**Code:**

```
// iter/iter.cpp
vector<int> counts{1,2,3,4,5,6};
vector<int>::iterator second = counts.begin()
    +1;
auto fourth = second+2;
counts.erase(second, fourth);
cout << counts[0]
    << "," << counts[1] << '\n';
```

**Output:**

```
[iter] erase2:
1, 4
```

(Also erasing a single element without end iterator.)

The `insert` operation takes a target iterator after which the insertion takes place, and two iterators for the range that will be inserted. This will extend the size of the target container.

Insert at iterator: value, single iterator, or range:

**Code:**

```
// iter/iter.cpp
vector<int> counts{1,2,3,4,5,6},
zeros{0,0};
auto after_one = zeros.begin() +1;
zeros.insert
( after_one,
  counts.begin() +1,
  counts.begin() +3 );
cout << zeros[0] << ","
    << zeros[1] << ","
    << zeros[2] << ","
    << zeros[3]
    << '\n';
```

**Output:**

```
[iter] insert2:
0,2,3,0
```

#### 14.2.3.1 Indexing and iterating

Functions that would return an array element or location, now return iterators. For instance:

- `find` returns an iterator pointing to the first element equal to the value we are finding;
- `max_element` returns an iterator pointing to the element with maximum value.

One of the arguments for range-based indexing was that we get a simple syntax if we don't need the index. Is it possible to use iterators and still get the index? Yes, that's what the function `distance` is for.

Find ‘index’ by getting the distance between two iterators:

**Code:**

```
// loop/distance.cpp
vector<int> numbers{1,3,5,7,9};
auto it=numbers.begin();
while ( it!=numbers.end() ) {
    auto d = distance(numbers.begin(),it);
    cout << "At distance " << d
        << ":" << *it << '\n';
    ++it;
}
```

**Output:**

```
[loop] distance:
At distance 0: 1
At distance 1: 3
At distance 2: 5
At distance 3: 7
At distance 4: 9
```

**Exercise 14.4.** Use the above vector methods to return, given a `std::vector<float>`, the integer index of its maximum element.

#### 14.2.4 Forming sub-arrays

There is a `std::vector` constructor that uses iterators. This can for instance be used to create a *subvector*. In the simplest case, you would make a copy of a vector using begin/end iterators:

```
vector<int> sub( othervec.begin(),othervec.end() );
```

Note that the subvector is formed as a copy of the original elements. Vectors completely ‘own’ their elements. For non-owning subvectors you would need `span`; section 10.8.3.

Some more examples. We form a subvector:

**Code:**

```
// iter/iter.cpp
vector<int> vec{11,22,33,44,55,66};
auto second = vec.begin(); ++second;
auto before = vec.end(); before--;
vector<int> sub(vec.data()+1,vec.data()+vec.
    size()-1);
cout << "no first and last: ";
for ( auto i : sub ) cout << i << ", ";
cout << '\n';
/* ... */
vector<int> vec{11,22,33,44,55,66};
auto second = vec.begin(); ++second;
auto before = vec.end(); before--;
// vector<int> sub(second,before);
vector<int> sub; sub.assign(second,before);
cout << "vector at " << (long)vec.data() <<
    '\n';
cout << "sub at " << (long)sub.data() << '\n'
    ';

cout << "no first and last: ";
for ( auto i : sub ) cout << i << ", ";
cout << '\n';
vec.at(1) = 222;
```

**Output:**

```
[iter] subvectorcopy:
no first and last: 22, 33,
→44, 55,
```

To demonstrate that the subvector is really a new object, not a subset of the original vector:

**Code:**

```
// iter/iter.cpp
vec.at(1) = 222;
cout << "did we get a change in the sub
vector? "
<< sub.at(0) << '\n';
```

**Output:**

```
[iter] subvectornew:
did we get a change in the
→sub vector? 22
```

### 14.3 Algorithms on ranges

Many simple algorithms on arrays, such as testing ‘there is’ or ‘for all’, no longer have to be coded out in C++. They can now be done with a single function from the `std::algorithm` library. This contains components that C++ programs may use to perform algorithmic operations on containers and other sequences.

So, even if you have learned a to code a specific algorithm yourself in the foregoing, you should study the following algorithms, or at least known that such algorithms exist. It’s what distinguishes a novice programmer from an industrial-grade (for want of a better term) programmer.

#### 14.3.1 Test Any/all

First we look at some algorithms that apply a predicate to the elements.

- Test if any element satisfies a condition: `any_of`. Note that both `std::any_of` and `std::ranges::any_of` exist. The former uses explicit iterators; we are mostly interested the latter, which takes a rangable object. The same distinction holds for the next two algorithms.
- Test if all elements satisfy a condition: `all_of`.
- Test if no elements satisfy a condition: `none_of`.
- Apply an operation to all elements: `for_each`.

**Remark** These tests correspond to mathematical quantors, and can be helpful in writing provably correct code. See section ??.

The object to which the function applies is not specified directly; rather, you have to specify a start and end iterator.

As an example of applying a predicate we look at a couple of examples of using `any_of`. This returns true or false depending on whether the predicate is true for any element of the container. Such boolean range algorithms use *short-circuit evaluation*: they stop evaluating range elements at the first true (for ‘there is’) or false (for ‘for all’).

Reduction with boolean result:  
See if any element satisfies a test

**Code:**

```
// iter/eachr.cpp
#include <ranges>
#include <algorithm>
/* ... */
vector<int> ints{1,2,3,4,5,7,8,13,14};
std::ranges::for_each
( ints,
  [] ( int i ) -> void {
    cout << i << '\n';
}
);
```

**Output:**  
[iter] eachr:  
1  
2  
3  
4  
5  
7  
8  
13  
14

(Why wouldn't you use a `accumulate` reduction?)

Here is an example using `any_of` to find whether a certain element appears in a vector:

**Code:**

```
// iter/eachr.cpp
vector<int> ints{1,2,3,4,5,7,8,13,14};
bool there_was_an_8 =
  std::ranges::any_of
( ints,
  [] ( int i ) -> bool {
    return i==8;
}
);
cout << "There was an 8: " << boolalpha <<
there_was_an_8 << '\n';
```

**Output:**  
[iter] anyr:  
There was an 8: true

Capturing the value to be tested gives:

**Code:**

```
// iter/eachr.cpp
vector<int> ints{1,2,3,4,5,7,8,13,14};
int tofind = 8;
bool there_was_an_8 =
  std::ranges::any_of
( ints,
  [tofind] ( int i ) -> bool {
    return i==tofind;
}
);
cout << "There was an 8: " << boolalpha <<
there_was_an_8 << '\n';
```

**Output:**  
[iter] anyc:  
There was an 8: true

**Remark** The previous examples rely on C++20 ranges. With iterators they look like this:

**Code:**

```
// iter/each.cpp
vector<int> ints{2,3,4,5,7,8,13,14,15};
bool there_was_an_8 =
    any_of( ints.begin(), ints.end(),
        [] ( int i ) -> bool {
            return i==8;
        }
    );
cout << "There was an 8: " << boolalpha <<
    there_was_an_8 << '\n';
```

**Output:**  
**[iter] any:**  
*There was an 8: true*

**Code:**

```
// iter/each.cpp
#include <algorithm>
/* ... */
vector<int> ints{2,3,4,5,7,8,13,14,15};
for_each( ints.begin(), ints.end(),
    [] ( int i ) -> void {
        cout << i << '\n';
    }
);
```

**Output:**  
**[iter] each:**  
*2  
3  
4  
5  
7  
8  
13  
14  
15*

### 14.3.2 Apply to each

The **for\_each** algorithm applies a function to every element of a container. Unlike the previous algorithms, this can alter the elements.

To introduce the syntax, we look at the pointless example of outputting each element:

Apply **cout** to each array element:

**Code:**

```
// iter/eachr.cpp
#include <ranges>
#include <algorithm>
/* ... */
vector<int> ints{1,2,3,4,5,7,8,13,14};
std::ranges::for_each
( ints,
    [] ( int i ) -> void {
        cout << i << '\n';
    }
);
```

**Output:**  
**[iter] eachr:**  
*1  
2  
3  
4  
5  
7  
8  
13  
14*

**Exercise 14.5.** Use `for_each` to sum the elements of a vector.

Hint: the problem is how to treat the sum variable. Do not use a global variable!

### 14.3.3 Iterator result

Some algorithms do not result in a value, but rather in an iterator that points to the location of that value. Examples: `min_element` takes a begin and end iterator, and returns the iterator in between where the minimum element is found. To find the actual value, we need to ‘dereference’ the iterator:

```
// range/minelt.cpp
vector<float> elements{.5f, 1.f, 1.5f};
auto min_iter = std::min_element
    (elements.begin(), elements.end());
cout << "Min: " << *min_iter << '\n';
```

Similarly `max_element`.

### 14.3.4 Mapping

The `transform` algorithm applies a function to each container element and returning the result. There are two variants. Using `std::transform` and iterators, you can write the result into a second container:

**Code:**

```
// rangestd/range.cpp
vector<float> in{1, 2, 3, 4}, out(4);
std::transform
( in.begin(), in.end(), out.begin(),
[] (float x) { return std::sqrt(x); } )
;
for ( auto x : out ) cout << x << '\n';
```

**Output:**  
**[rangestd] transformi:**

```
1
1.41421
1.73205
2
```

Using `ranges::transform` you obtain another range view:

**Code:**

```
// rangestd/range.cpp
vector<float> in{1, 2, 3, 4}, out(4);
for ( auto x : in
    | rng::views::transform
        ( [] (float x) { return std::sqrt
            (x); } ) )
    cout << x << '\n';
```

**Output:**  
**[rangestd] transformr:**

```
1
1.41421
1.73205
2
```

### 14.3.5 Reduction

Numerical *reductions* can be applied `accumulate` (using iterators) or `ranges::accumulate` (using ranges), both requiring the `numeric` header. If no reduction operator is specified, a *sum reduction* is performed.

Default is sum reduction:

```
// range/sumsquare.cpp
vector<float> elements{.5f, 1.f, 1.5f};
auto sumsq =
    rng::accumulate
    (elements
     | rng::views::transform( [] (auto e) { return e*e; } ),
     0.f );
cout << "Sum of squares: " << sumsq << '\n';
```

Other binary arithmetic operators that can be used as *reduction operator* are found in [functional](#):

- `plus`, `minus`, `multipplies`, `divides`,
- integers only: `modulus`
- boolean: `logical_and`, `logical_or`

This header also contains the unary `negate` operator, which can of course not be used for reductions.

As an example of an explicitly specified reduction operator:

```
auto p = std::accumulate
( x.begin(), x.end(), 1.f,
  std::multiplies<float>()
);
```

Note:

- that the operator is templated, and that it is followed by parentheses to become a functor, rather than a class;
- that the accumulate function is templated, and it takes its type from the init value. Thus, in the above example, a value of `1` would have turned this into an integer operation.

Supply multiply operator:

**Code:**

```
// range/reduce.cpp
vector<int> v{1,3,5,7};
auto product =
    rng::accumulate
    (v, 2, multiplies<>());
cout << "multiplied: " << product << '\n';
```

**Output:**

```
[range] product:
multiplied: 210
```

Specific for the max reduction is `max_element`. This can be called without a comparator (for numerical max), or with a comparator for general maximum operations. The maximum and minimum algorithms return an iterator, rather than only the max/min value.

Example: maximum relative deviation from a quantity:

```
max_element(myvalue.begin(), myvalue.end(),
[my_sum_of_squares] (double x, double y) -> bool {
    return std::abs( (my_sum_of_squares-x) / x ) < std::abs( (my_sum_of_squares-y) /
y );
})
```

```
) ;
```

For more complicated lambdas used in `accumulate`,

- the first argument should be the reduce type,
- the second argument should be the iterated type

In the following example we accumulate one member of a class:

Two-component class and accumulate just one:

Class with `i`, `j`:

```
// stl/reduce.cpp
class x {
public:
    int i, j;
    x() {};
    x(int i, int j) : i(i), j(j) {};
};
```

Accumulate one component:

```
// stl/reduce.cpp
std::vector<x> xs(5);
auto xxx =
    std::accumulate(
        xs.begin(), xs.end(), 0,
        [] ( int init, x x1 ) -> int {
            return x1.i+init; }
    );
```

**Remark** The `accumulate` algorithm does not have a parallel version with an execution policy. For that, see `std::reduce` in the `numeric` header.

### 14.3.6 Sorting

The `algorithm` header also has a function `sort`.

With iterators you can easily apply this to things such as vectors:

```
sort( myvec.begin(), myvec.end() );
```

The comparison used by default is ascending. You can specify other compare functions:

```
sort( myvec.begin(), myvec.end(),
      [] (int i, int) { return i>j; }
);
```

or

```
sort( people.begin(), people.end(),
      [] ( const Person& lhs, const Person& rhs ) {
          return lhs.name < rhs.name; }
)
```

With iterators you can also do things like sorting a part of the vector:

```
Code:
// rangestd/sort.cpp
vector<int> v{3,1,2,4,5,7,9,11,12,8,10};
cout << "Original vector: "
    << vector_as_string(v) << '\n';

auto v_std(v);
std::sort( v_std.begin(),v_std.begin() + 5 );
cout << "Five elements sorts: "
    << vector_as_string(v_std) << '\n';
```

```
Output:
[range] sortit:
Original vector: 3, 1, 2, 4,
              ↪5, 7, 9, 11, 12, 8, 10,
Five elements sorts: 1, 2,
              ↪3, 4, 5, 7, 9, 11, 12,
              ↪8, 10,
```

## 14.4 Parallel execution policies

The C++17 standard added the *ExecutionPolicy* concept to standard algorithms, describing how an element of the `algorithm` library may be executed in parallel.

There are three choices for the *execution policy*, defined in the `execution` header:

- `std::execution::seq`: iterations may not be parallelized, but are indeterminately sequenced in the evaluation thread.
- `std::execution::par`: iterations may be executed in parallel, but are indetermined sequences;
- `std::execution::par_unseq`: iterations may be parallelized, vectorized, migrated over threads.
- `std::execution::unseq` (since C++20): iterations are allowed to be vectorized over.

To accomodate the indeterminate evaluation order, new ‘unordered’ algorithms are introduced based on existing ‘ordered algorithms’:

- `reduce`: similar `accumulate`, but unordered and therefore parallelizable through an *ExecutionPolicy*;
  - `inclusive_scan`: similar to `partial_sum`;
  - `exclusive_scan`: no ordered equivalent
  - `transform_reduce`, `transform_inclusive_scan`, `transform_exclusive_scan`. Example usage:
- ```
transform_reduce( execpol, it_first,it_last, init_val, reduce_op,
                 transform_op );
```

**Remark** The practical implementation of execution policies is often done through the Threading Building Blocks (Intel) (TBB) library. This means that you may need to reflect that in your compiler and link flags. It also means that you can control the amount of parallelism through setting the number of TBB threads:

```
#include "tbb/tbb.h"
// ...
tbb::global_control ctl(tbb::global_control::max_allowed_parallelism, nthreads);
```

Some performance measurements on these are given in MPI/OpenMP book [12], section 19.4.2.

## 14.5 Classification of algorithms

(Taken from a lecture by Dietmar Kühl at CppCon 2017, as per its copyright notice.)

### 14.5.1 Non-parallel algorithms

#### 14.5.1.1 Order $O(1)$ algorithms

`clamp, destroy_at, gcd, iter_swap, lcm, max, min, minmax,`

#### 14.5.1.2 Order $O(\log n)$ algorithms

`binary_search, equal_range, lower_bound, partition_point, pop_heap, push_heap, upperbound,`

#### 14.5.1.3 Heap algorithms

`make_heap, sort_heap,`

#### 14.5.1.4 Permutation algorithms

`is_permutation, next_permutation, prev_permutation,`

#### 14.5.1.5 Overlapping algorithms

`copy_backward, move_backward,`

#### 14.5.1.6 Renamed algorithms

`accumulate, partial_sum,`

#### 14.5.1.7 Oddball algorithms

`iota, sample, shuffle,`

## 14.5.2 Parallel algorithms

### 14.5.2.1 Map algorithms

`copy, copy_n, destroy, destroy_n, fill, fill_n, for_each, for_each_n, generate, generate_n, move, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, swap_ranges, transform, uninitialized_*,`

### 14.5.2.2 Reduce algorithms

`adjacent_find, all_of, any_of, count, count_if, equal, find, find_end, find_first_of, find_if, find_if_not, includes, inner_product, is_heap, is_heap_until, is_partitioned, is_sorted, is_sorted_until, lexicographical_compare, max_element, min_element, minmax_element, mismatch, none_of, reduce, search, search_n,`

#### 14.5.2.3 Scan algorithms

`exclusive_scan, inclusive_scan,`

#### 14.5.2.4 Fused algorithms

`transform_exclusive_scan, transform_inclusive_scan, transform_reduce,`

#### 14.5.2.5 Gather algorithms

`copy_if, partition_copy, remove, remove_copy, remove_copy_if, remove_if, rotate, unique, unique_copy,`

#### 14.5.2.6 Special algorithms

`adjacent_difference, inplace_merge, merge, nth_element, partial_sort, partial_sort_copy, partition, rotate, set_difference, set_intersection, set_symmetric_difference, set_union, sort, stable_partition, stable_sort,`

## 14.6 Advanced topics

### 14.6.1 Utilities

Peek into a pipeline:

```
rng::whatever | emit | rng::whatever
```

with

```
auto emit = [] ( const char delim=‘ ’ ) {
    return rng::views::filter
    ( [delim] ( auto&& value ) noexcept {
        std::cout << value << delim; return true; } )
};
```

Courtesy of Ruyard Merriam <https://www.youtube.com/watch?v=h1aRKYs-FqA>

### 14.6.2 Range types

Types of ranges:

- `std::ranges::input_range` : iterate forward at least once, as if you’re accepting input with `cin` and such.
- `std::ranges::forward_range` : can be iterated forward, (for instance with plus-plus), multiple times, as in a *single-linked list*.
- `std::ranges::bidirectional_range` : can be iterated in both directions, for instance with plus-plus and minus-minus.
- `std::ranges::random_access_range` items can be found in constant time, such as with square bracket indexing.
- `std::ranges::contiguous_range` : items are stored consecutively in memory, making address calculations possible.

### 14.6.3 Make your own iterator

You know that you can iterate over `vector` objects:

```
vector<float> myvector(20);
float s{0.f};
for ( auto copy_of_float : myvector )
    s += copy_of_float;
for ( auto &ref_to_float : myvector )
    ref_to_float /= s;
```

(Many other standard library classes are iterable like this.)

This range-based syntax was introduced in C++11; prior to that, looping over a vector looked like:

```
for (auto elt_ptr=vec.begin(); elt_ptr!=vec.end(); ++elt_ptr)
    element = *elt_ptr;
```

The more modern syntax is translated into this, and from this we see that the requirements for a class to be *iterable* are the existence of:

- `begin` and `end` methods; these give an *iterator* object in the initial and final state respectively;
- a test on whether the iterator has reached the final state;
- an *increment operator*; and
- a ‘star’ operator (that is sort of, but not quite, like a *dereference*) that gives the value that the iterator points at.

Some remarks:

- This is one of the few places where you need the asterisk in C++. However, you’re applying it to an iterator, not a pointer, and this is an operator you are applying (see section 9.5.7).
- As with a normal loop, the `end` iterator points just beyond the end of the vector.
- You can do ‘pointer arithmetic’ on iterators (section ??), as you can see in the `++elt_ptr` update part of the loop header.



# Chapter 15

## References

### 15.1 Reference

This section contains further facts about references, which you have already seen as a mechanism for parameter passing; section 7.5.2. Make sure you study that material first.

Passing a variable to a routine copies the value; in the routine, the variable is local.

```
Code:  
// func/localparm.cpp  
void change_scalar(int i) {  
    i += 1;  
}  
/* ... */  
number = 3;  
cout << "Number is 3: "  
     << number << '\n';  
change_scalar(number);  
cout << "is it still 3? Let's see: "  
     << number << '\n';
```

```
Output:  
[func] localparm:  
Number is 3: 3  
is it still 3? Let's see: 3
```

If you do want to make the change visible in the *calling environment*, use a reference:

```
// basic/arraypass.cpp  
void change_scalar_by_reference(int &i) { i += 1; }
```

This requires no change to the calling program. (Some people who are used to C dislike this, since you can not see from the use of a function whether it passes *by reference* or *by value*.)

### 15.2 Pass by reference

If you use a mathematical style of subprograms, where some values go in, and a new entity comes out, in effect all the inputs can be copied. This style is called *functional programming*, and it makes it possible for the compiler to reason about your program. The only thing you have to worry about is the cost of copying, if the inputs are of non-trivial size, such as arrays.

## 15. References

---

However, sometimes you want to alter the inputs, so instead of a copy you need a way of accessing the actual input object. That's what *references* are invented for: to allow a subprogram access to the actual input entity.

A bonus of using references is that you do not incur the cost of copying. So what if you want this efficiency, but your program is really functional in design? Then you can use a *const reference*: the argument is passed by reference, but you indicate explicitly that the subprogram does not alter it, again allowing compiler optimizations.

A reference makes the function parameter a synonym of the argument.

```
void f( int &i ) { i += 1; }
int main() {
    int i = 2;
    f(i); // makes it 3
```

Passing a big object without copying:

```
class BigDude {
public:
    vector<double> array(5000000);
}

void f(BigDude d) {
    cout << d.array[0];
}

int main() {
    BigDude big;
    f(big); // whole thing is copied
```

Instead write:

```
void f( BigDude &thing ) { .... };
```

Prevent changes:

```
void f( const BigDude &thing ) { .... };
```

### 15.3 Reference to class members

Here is the naive way of returning a class member:

```
class Object {
private:
    SomeType thing;
public:
    SomeType get_thing() {
        return thing; }
};
```

Now the return statement makes a copy of *thing*, which may be the desired behavior, and it may not be. If you don't need an actual copy, but you want access to the actual data member, you can return the member by *reference*:

```
SomeType &get_thing() {
    return thing; }
```

Now you have write access to an internal (maybe private!) data member. You may want to have that, but if you don't use a *const reference*:

```
Code:
// const/constref.cpp
class has_int {
private:
    int mine{1};
public:
    const int& int_to_get() { return mine; }
    int& int_to_set() { return mine; }
    void inc() { ++mine; }
};

/* ...
has_int an_int;
an_int.inc(); an_int.inc();
printf("Contained int is now: {}",
       an_int.int_to_get());
/* Compiler error: an_int.int_to_get() = 5;
*/
an_int.int_to_set() = 17;
printf("Contained int is now: {}",
       an_int.int_to_get());
```

```
Output:
[const] constref:
Contained int is now: 4
Contained int is now: 17
```

In the above example, the function giving a reference was used in the left-hand side of an assignment. If you would use it on the right-hand side, you would not get a reference.

Let's again make a class where we can get a reference to the internals:

```
// func/rhsref.cpp
class myclass {
private:
    int stored{0};
public:
    myclass(int i) : stored(i) {};
    int &data() { return stored; };
};
```

Now we explore various ways of using that reference on the right-hand side:

```
Code:
// func/rhsref.cpp
myclass obj(5);
println("object data: {}", obj.data());
int dcopy = obj.data();
++dcopy;
println("object data: {}", obj.data());
int &dref = obj.data();
++dref;
println("object data: {}", obj.data());
auto dauto = obj.data();
++dauto;
println("object data: {}", obj.data());
auto &aref = obj.data();
++aref;
println("object data: {}", obj.data());
```

```
Output:
[func] rhsref:
object data: 5
object data: 5
object data: 6
object data: 6
object data: 7
```

You see that, despite the fact that the method `data` was defined as returning a reference, you still need to indicate whether the left-hand side is a reference.

## 15.4 Reference to array members

You can define various operator, such as `+-* /` arithmetic operators, to act on classes, with your own provided implementation; see section 9.5.7. You can also define the parentheses and square brackets operators, so make your object look like a function or an array respectively.

These mechanisms can also be used to provide safe access to arrays and/or vectors that are private to the object.

Suppose you have an object that contains an `int` array. You can return an element by defining the subscript (square bracket) operator for the class:

```
// array/getindex1.cpp
class vector10 {
private:
    int array[10];
public:
    /* ... */
    int operator()(int i) {
        return array[i];
    };
    int operator[](int i) {
        return array[i];
    };
};
/* ... */
vector10 v;
cout << v(3) << '\n';
cout << v[2] << '\n';
/* compilation error: v(3) = -2; */
```

Note that `return array[i]` will return a copy of the array element, so it is not possible to write

```
myobject[5] = 6;
```

For this we need to return a reference to the array element:

```
// array/getindex2.cpp
int& operator[](int i) {
    return array[i];
}
/* ...
cout << v[2] << '\n';
v[2] = -2;
cout << v[2] << '\n';
```

Your reason for wanting to return a reference could be to prevent the *copy of the return result* that is induced by the `return` statement. In this case, you may not want to be able to alter the object contents, so you can return a *const reference*:

```
// array/getindex3.cpp
const int& operator[](int i) {
    return array[i];
}
/* ...
cout << v[2] << '\n';
/* compilation error: v[2] = -2; */
```

## 15.5 Addresses

C programmers are accustomed to using the ampersand for getting the address of variables, and in C++: objects. C++ programmers need to beware that it's possible to redefine the ampersand:

```
T operator&() { /* stuff */ };
```

If you absolutely need an address, use the `addressof` function.



# Chapter 16

## Pointers

Pointers are an indirect way of associating an entity with a variable name. Consider for instance the circular-sounding definition of a list:

A list consists of nodes, where a node contains some information in its ‘head’, and the node has a ‘tail’ that is a list.

Naive code:

```
class Node {  
private:  
    int value;  
    Node tail; // This does not work!  
    /* ... */  
};
```

This does not work: would take infinite memory.

Indirect inclusion: only ‘point’ to the tail:

```
class Node {  
private:  
    int value;  
    PointToNode tail;  
    /* ... */  
};
```

This chapter will explain C++ *smart pointers*, and give some uses for them.

### 16.1 Pointer usage

Initially, we will focus on pointers to objects of some class.

Simple class that stores one number:

## 16. Pointers

---

Definition:

```
// pointer/pointx.cpp
class HasX {
private:
    double x;
public:
    HasX( double x ) : x(x) {};
    auto value() { return x; };
    void set(double xx) {
        x = xx;
    };
};
```

Example usage

```
// pointer/pointx.cpp
HasX xobj(5);
printf("%{}", xobj.value());
xobj.set(6);
printf("%{}", xobj.value());
```

With class objects, the ‘dot’ notation for class members becomes an ‘arrow’ notation when you use a pointer. If you have an object *obj* *x* with a member *y*, you access that with *x.y*; if you have a pointer *x* to such an object, you write *x->y*.

- If *x* is object with member *y*:  
*x.y*
- If *xx* is pointer to object with member *y*:  
*xx->y*
- Arrow notation works with C-style pointers and new shared/unique pointers.

Instead of creating an object, you now create an object with a pointer to it, in a single call.

Note that you don’t first create an object, and then set a pointer to it, the way it happens in many other languages. Smart pointers work differently: you create the object and the pointer to it in one call.

```
make_shared<ClassName>( constructor arguments );
```

The resulting object is of type `shared_ptr<ClassName>`, but you can save yourself spelling that out, and use `auto` instead.

Object vs pointed-object:

Code:

```
// pointer/pointx.cpp
#include <memory>
using std::make_shared;

/* ... */
HasX xobj(5);
printf("%{}", xobj.value());
xobj.set(6);
printf("%{}", xobj.value());

auto xptr = make_shared<HasX>(5);
printf("%{}", xptr->value());
xptr->set(6);
printf("%{}", xptr->value());
```

Output:

[pointer] pointx:

```
5
6
5
6
```

Using smart pointers requires at the top of your file:

```
#include <memory>
using std::shared_ptr;
using std::make_shared;

using std::unique_ptr;
using std::make_unique;
```

Why do we use pointers? Pointers are generally said to pertain to ‘ownership’, and with pointers we can let one object be ‘owned’ equally by two variables.

Set two pointers to the same object:

**Code:**

```
// pointer/twopoint.cpp
auto xptr = make_shared<HasX>(5);
auto yptr = xptr;
cout << xptr->get() << '\n';
yptr->set(6);
cout << xptr->get() << '\n';
```

**Output:**  
[pointer] twopoint:  
5  
6

Sometimes you want to get hold of the actual object being pointed to. For this you can use the `operator*` which dereferences the pointer.

Example: function

```
float distance_to_origin( Point p );
```

How do you apply that to a `shared_ptr<Point>`?

```
shared_ptr<Point> p;
distance_to_origin( *p );
```

So far, you have created a smart pointer variable by immediately initializing it. This is good C++ practice. However, sometimes you need to declare such a variable and you can not immediately initialize it. What value does that variable have? Well, that’s undefined, so it’s a good idea to initialize it to `nullptr`, a value that can be tested.

Initialize smart pointer to null pointer; test on null value:

```
shared_ptr<Foo> foo_ptr; // is nullptr by default initialization;
// stuff
if (foo_ptr!=nullptr)
    foo_ptr->do_something();
```

**Exercise 16.1.** If you are doing the geometry project (chapter 46) you can now do exercises 46.20 and 46.22.

## 16.2 How smart pointers prevent memory problems

Smart pointers (and the way C++ deals with containers) help prevent some memory problems that are common in C code. We discuss the major types here.

### 16.2.1 Memory leaks

One problem with C-style pointers is the chance of a *memory leak*. If a pointer to a block of memory goes out of scope, the block is not returned to the OS, but it is no longer accessible.

```
// attach memory to 'array':  
double *array = new double[25];  
// do something with array  
// overwrite the pointer  
array = new double[26];  
// the first allocated memory is still reserved.
```

Shared and unique pointers do not have this problem: if they go out of scope, or are overwritten, the destructor on the object is called, thereby releasing any allocated memory.

An example.

We need a class with constructor and destructor tracing:

```
// pointer/ptr1.cpp  
class thing {  
public:  
    thing() { cout << "... calling constructor\n"; }  
    ~thing() { cout << "... calling destructor\n"; }  
};
```

Just to illustrate that the destructor gets called when the object goes out of scope we show an example without any pointers for now:

**Code:**

```
// pointer/ptr0.cpp  
cout << "Outside\n";  
{  
    thing x;  
    cout << "create done\n";  
}  
cout << "back outside\n";
```

**Output:**

```
[pointer] ptr0:  
Outside  
.. calling constructor  
create done  
.. calling destructor  
back outside
```

Now we do the same with a pointer, to illustrate that the destructor of the object is called when the pointer no longer points to the object. We do this by assigning `nullptr` to the pointer. (This is very different from `NULL` in C: the null pointer is actually an object with a type; see section 16.3.6.)

Let's create a pointer and overwrite it:

<b>Code:</b> <pre>// pointer/ptr1.cpp cout &lt;&lt; "set pointer1"     &lt;&lt; '\n'; auto thing_ptr1 =     make_shared&lt;thing&gt;(); cout &lt;&lt; "overwrite pointer"     &lt;&lt; '\n'; thing_ptr1 = nullptr;</pre>	<b>Output:</b> [pointer] ptr1: set pointer1 .. calling constructor overwrite pointer .. calling destructor
---	---

However, if a pointer is copied, there are two pointers to the same block of memory, and only when both disappear, or point elsewhere, is the object deallocated.

<b>Code:</b> <pre>// pointer/ptr2.cpp cout &lt;&lt; "set pointer2" &lt;&lt; '\n'; auto thing_ptr2 =     make_shared&lt;thing&gt;(); cout &lt;&lt; "set pointer3 by copy"     &lt;&lt; '\n'; auto thing_ptr3 = thing_ptr2; cout &lt;&lt; "overwrite pointer2"     &lt;&lt; '\n'; thing_ptr2 = nullptr; cout &lt;&lt; "overwrite pointer3"     &lt;&lt; '\n'; thing_ptr3 = nullptr;</pre>	<b>Output:</b> [pointer] ptr2: set pointer2 .. calling constructor set pointer3 by copy overwrite pointer2 overwrite pointer3 .. calling destructor
--	--

- The object counts how many pointers there are:
- ‘reference counting’
- A pointed-to object is deallocated if no one points to it.

### 16.2.2 Memory leaks and exceptions

A more obscure source of memory leaks has to do with exceptions:

```
void f() {
    double *x = new double[50];
    throw("something");
    delete x;
}
```

Because of the exception (which can of course come from a nested function call) the `delete` statement is never reached, and the allocated memory is leaked. Smart pointers solve this problem: throwing the exception leaves the scope and therefore the destructor is called.

### 16.2.3 Use after free

A serious memory problem arises from using a pointer after it has been free'd, the *use-after-free* condition:

```
double *x = new double[N];
/* operations on x */
delete[] x;
... x[500] ... // use of free'd memory
```

By the time you access the freed memory that memory may have been allocated to, and used by, another array. Your program in other words may continue to run, but with unintended values. On the other hand, a smart pointer will have been set to `nullptr`, and dereferencing that will likely crash your code.

## 16.3 Advanced topics

### 16.3.1 Get the pointed data

Most of the time, accessing the target of the pointer through the arrow notation is enough. However, if you actually want the object, you can get it with `get`.

```
X->y;
// is the same as
X.get() ->y;
// is the same as
( *X.get() ) .y;
```

Note that this does not give you the pointed object, but a traditional pointer.

Demonstrate the `get` function:

**Code :**

```
// pointer/poinky.cpp
auto Y = make_shared<HasY>(5);
cout << Y->y << '\n';
Y.get() ->y = 6;
cout << ( *Y.get() ) .y << '\n' ;
```

**Output :**  
**[pointer] poinky:**  
5  
6

**Remark** Using the `get` method has the (small) advantage that it still works if the `->` operator is overloaded.

### 16.3.2 Unique pointers

Shared pointers are fairly easy to program, and they come with lots of advantages, such as the automatic memory management. However, they have more overhead than strictly necessary because they have a *reference count* mechanism to support the memory management. Therefore, there exists a *unique pointer*, `unique_ptr`, for cases where an object will only ever be ‘owned’ by one pointer.

While unique pointers may have lower overhead, they are more tricky to use, for instance because you can not copy them.

---

```
auto X = make_unique< VectorField >();
for ( /* many time steps */ ) {
    X->update();
```

### 16.3.3 Base and derived pointers

Suppose you have base and derived classes:

```
class A {};
class B : public A {};
```

Just like you could assign a `B` object to an `A` variable:

```
B b_object;
A a_object = b_object;
```

is it possible to assign a `B` pointer to an `A` pointer?

The following construct makes this possible:

```
auto a_ptr = shared_ptr<A>( make_shared<B>() );
```

See also section [26.2.2](#) about casting between base and derived pointers and references.

### 16.3.4 Shared pointer to ‘this’

Inside an object method, the object is accessible as `this`. This is a pointer in the classical sense. So what if you want to refer to ‘this’ but you need a shared pointer?

For instance, suppose you’re writing a linked list code, and your `node` class has a method `prepend_or_append` that gives a shared pointer to the new head of the list.

Your code would start something like this, handling the case where the new node is appended to the current:

```
shared_pointer<node> node::prepend_or_append
    ( shared_ptr<node> other ) {
    if (other->value>this->value) {
        this->tail = other;
        return *this; // ALMOST BUT NOT QUITE
    } else {
        other->tail = this; // ALSO A LITTLE OFF
        return other;
    }
}
```

Now you need to return this node, as a shared pointer. However, `this` is a `node*`, not a `shared_ptr<node>`.

The solution here is that you can return

```
return this->shared_from_this();
```

if you have defined your node class to inherit from what probably looks like magic:

```
class node : public enable_shared_from_this<node>
```

Note that you can only return a `shared_from_this` if already a valid shared pointer to that object exists.

### 16.3.5 Weak pointers

In addition to shared and unique pointers, which own an object, there is also `weak_ptr` which creates a *weak pointer*. This pointer type does not own, but at least it knows when it dangles.

Unlike unique and shared pointers, a weak pointer can not immediately give its pointed value; it first needs to convert to a shared pointer with the `lock` method.

There is a subtlety with weak pointers and shared pointers. The call

```
auto sp = shared_ptr<Obj>( new Obj );
```

creates first the object, then the ‘control block’ that counts owners. On the other hand,

```
auto sp = make_shared<Obj>();
```

does a single allocation for object and control block.

### 16.3.6 Null pointer

In C there was a macro `NULL` that, only by convention, was used to indicate *null pointers*: pointers that do not point to anything. C++ has the `nullptr`, which is an object of type `std::nullptr_t`.

There is also the abstract concept of ‘the null pointer value of a type’. Both zero and `NULL` can be converted to that, but use of `nullptr` is preferred.

There are some scenarios where this is useful, for instance, with polymorphic functions:

```
void f(int);  
void f(int*);
```

Calling `f(ptr)` where the pointer is `NULL`, the first function is called, whereas with `nullptr` the second is called.

Note: dereferencing a `nullptr` does not give an exception, so you need to test explicitly on null pointers.

### 16.3.7 Pointers to non-objects

In the introduction to this chapter we argued that many of the uses for pointers that existed in C have gone away in C++, and the main one left is the case where multiple objects share ‘ownership’ of some other object. You can make shared pointers to scalar data, for instance to an array of scalars. You then get the advantage of the memory management, but you do not get the `size` function and such that you would have if you’d used a `vector` object.

Here is an example of a pointer to a solitary double:

<b>Code:</b> <pre>// pointer/ptrdouble.cpp // shared pointer to allocated double auto array = shared_ptr&lt;double&gt;( new double     ); double *ptr = array.get(); array.get()[0] = 2.; cout &lt;&lt; ptr[0] &lt;&lt; '\n';</pre>	<b>Output:</b> [pointer] ptrdouble: 2
--	---

It is possible to initialize that double:

```
Code:
// pointer/ptrdouble.cpp
// shared pointer to initialized double
auto array = make_shared<double>(50);
double *ptr = array.get();
cout << ptr[0] << '\n';
```

```
Output:
[pointer] ptrdoubleinit:
50
```

You can also have pointers to arrays:

```
Code:
// pointer/ptrdouble.cpp
auto array = std::make_unique<double[]>(50);
double *ptr = array.get();
array[1] = 2.81;
cout << ptr[1] << '\n';
```

```
Output:
[pointer] ptrdoublearray:
2.81
```

The constructor syntax is a little involved for vectors:

```
auto x = make_shared<vector<double>>(vector<double>{1.1, 2.2});
```

## 16.4 Smart pointers vs C pointers

We remark that there is less need for pointers in C++ than there was in C.

- To pass an argument *by reference*, use a *reference*. Section 7.5.
- Strings are done through `std::string`, not character arrays; see chapter 11.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see chapter 10.
- Traversing arrays and vectors can be done with ranges; section 10.2.
- Anything that obeys a scope should be created through a *constructor*, rather than using `malloc`

### 16.4.1 Smart pointers versus C-style address pointers

The oldstyle `&y` address pointer can not be made smart:

```
// pointer/address.cpp
auto
p = shared_ptr<HasY>(&y);
p->y = 3;
cout << "Pointer's y: "
<< p->y << '\n';
```

gives:

## 16. Pointers

---

```
address(56325,0x7fff977cc380) malloc: *** error for object  
0x7f9eeb9caf08: pointer being freed was not allocated
```

Smart pointers are much better than old style pointers

```
Obj *X;  
*X = Obj( /* args */ );
```

There is a final way of creating a shared pointer where you cast an old-style `new` object to shared pointer

```
auto p = shared_ptr<Obj>( new Obj );
```

This is not the preferred mode of creation, but it can be useful in the case of *weak pointers*; section [16.3.5](#).

# Chapter 17

## C-style pointers and arrays

In preceding chapters you have learned about `std::vector` as the preferred way of storing array data, `std::string` for character data, parameter passing by value or reference, and smart pointers for such data structures as linked lists. All these mechanisms are handled in C with a different pointer mechanism.

This chapter discusses the C mechanism. Study it, in case you ever need to deal with C code, but don't use it in your C++ code!

### 17.1 What is a pointer

The term pointer is used to denote a reference to a quantity. The reason that people like to use C as high performance language is that pointers are actually memory addresses. So you're programming 'close to the bare metal' and are in far going control over what your program does. C++ also has pointers, but there are fewer uses for them than for C pointers: vectors and references have made many of the uses of C-style pointers obsolete.

### 17.2 Pointers and addresses, C style

You have learned about variables, and maybe you have a mental concept of variables as 'named memory locations'. That is not too far off: while you are in the (dynamic) scope of a variable, it corresponds to a fixed memory location.

**Exercise 17.1.** When does a variable not always correspond to the same location in memory?

There is a mechanism of finding the actual address of a variable: you prefix its name by an ampersand. This address is integer-valued, but its range is actually greater than of the `int` type.

If you have an `int i`, then `&i` is the address of `i`.

An address is a (long) integer, denoting a memory address. Usually it is rendered in *hexadecimal* notation.

## 17. C-style pointers and arrays

```
Code:  
// pointer/coutpoint.cpp  
int i;  
cout << "address of i, decimal: "  
     << (long)&i << '\n';  
cout << "address of i, hex : "  
     << std::hex << &i << '\n';
```

```
Output:  
[pointer] coutpoint:  
address of i, decimal:  
    ↳140732703427524  
address of i, hex :  
    ↳0x7ffee2cbcfc4
```

Using purely C:

```
Code:  
// pointer/printfpoint.cpp  
int i;  
printf("address of i: %ld\n",  
        (long)(&i));  
printf(" same in hex: %lx\n",  
        (long)(&i));
```

```
Output:  
[pointer] printfpoint:  
address of i: 140732690693076  
same in hex: 7ffee2097bd4
```

Note that this use of the ampersand is different from defining references; compare section [7.5.2](#). However, there is never a confusion which is which since they are syntactically different.

You could just print out the address of a variable, which is sometimes useful for debugging. If you want to store the address, you need to create a variable of the appropriate type. This is done by taking a type and affixing a star to it.

The type of ‘`&i`’ is `int*`, pronounced ‘int-star’, or more formally: ‘pointer-to-int’.

You can create variables of this type:

```
int i;  
int* addr = &i;  
// exactly the same:  
int *addr = &i;
```

Now `addr` contains the memory address of `i`.

Now if you have have a pointer that refers to an int:

```
int i;  
int *iaddr = &i;
```

you can use (for instance `print`) that pointer, which gives you the address of the variable. If you want the value of the variable that the pointer points to, you need to *dereference* it.

Using `*addr` ‘dereferences’ the pointer: gives the thing it points to; the value of what is in the memory location.

```
Code:
// pointer/cintpointer.cpp
int i;
int* addr = &i;
i = 5;
cout << *addr << '\n';
i = 6;
cout << *addr << '\n';
```

Output:  
[pointer] cintpointer:  
5  
6

int x = 6;

A memory diagram showing a single integer variable x at address 946. The value 6 is stored at this address. An arrow labeled "x" points to the value 6.

int y = x;

A memory diagram showing two integer variables x and y. Variable x is at address 946 with value 6. Variable y is at address 958 with value 6. Arrows labeled "x" and "y" point to their respective values.

int \*xx = &x;

A memory diagram showing a pointer xx pointing to variable x. The pointer xx is at address 982 and contains the address 946. Variable x is at address 946 with value 6. A curved arrow labeled "\*xx" points from the address 946 back to the pointer xx.

x = 8;

A memory diagram showing variable x with value 8 at address 946. The pointer xx still points to the original address 946, which now contains the value 6. A curved arrow labeled "\*xx" points from the address 946 back to the pointer xx.

- `addr` is the address of `i`.
- You set `i` to 5; nothing changes about `addr`. This has the effect of writing 5 in the memory location of `i`.
- The first `cout` line dereferences `addr`, that is, looks up what is in that memory location.
- Next you change `i` to 6, that is, you write 6 in its memory location.
- The second `cout` looks in the same memory location as before, and now finds 6.

The syntax for declaring a pointer-to-sometype allows for a small variation, which indicates the two way you can interpret such a declaration.

Equivalent:

- `int* addr`: `addr` is an int-star, or

- `int *addr`: `*addr` is an int.

The notion `int* addr` is equivalent to `int *addr`, and semantically they are also the same: you could say that `addr` is an int-star, or you could say that `*addr` is an int.

### 17.3 Arrays and pointers

In section 10.10 you saw the treatment of C-style arrays in C++. Examples such as:

```
// pointer/arraypass.cpp
void set_array( double *x, int size) {
    for (int i=0; i<size; ++i)
        x[i] = 1.41;
}
/* ... */
double array[5] = {11,22,33,44,55};
set_array(array,5);
cout << array[0] << "...." << array[4] << '\n';
```

show that, even though parameters are normally passed by value, that is through copying, array parameters can be altered. The reason for this is that there is no actual array type, and what is passed is a pointer to the first element of the array. So arrays are still passed by value, just not the ‘value of the array’, but the value of its location.

So you could pass an array like this:

```
// array/arraypass.cpp
void array_set_star( double *ar, int idx, double val) {
    ar[idx] = val;
}
/* ... */
array_set_star(array,2,4.2);
```

Array and memory locations are largely the same:

**Code:**

```
// pointer/arrayaddr.cpp
double array[5] = {11,22,33,44,55};
double *addr_of_second = &(array[1]);
cout << *addr_of_second << '\n';
array[1] = 7.77;
cout << *addr_of_second << '\n';
```

**Output:**

```
[pointer] arrayaddr:
22
7.77
```

When an array is passed to a function, it behaves as an address:

**Code:**

```
// pointer/arraypass.cpp
void set_array( double *x, int size) {
    for (int i=0; i<size; ++i)
        x[i] = 1.41;
}
/* ... */
double array[5] = {11,22,33,44,55};
set_array(array,5);
cout << array[0] << "...." << array[4] << '\n';
```

**Output:**  
[pointer] arraypass:  
1.41....1.41

Note that these arrays don't know their size, so you need to pass it.

There is a `sizeof` function but beware:

**Code:**

```
// c/carray.c
void stat_f( int stat[] ) {
    printf(.. in function: %lu\n",sizeof(stat))
    ;
}
//codesnippet
```

**Output:**  
[c] carraystat:  
carray.c:16:40: warning: sizeof  
 ↪on array function  
 ↪parameter will return  
 ↪size of 'int \*' instead  
 ↪of 'int []'  
 ↪[-Wsizeof-array-argument]  
printf(.. in function:  
 ↪%lu\n",sizeof(stat));  
  
 ↪  
carray.c:15:18: note: declared  
 ↪here  
void stat\_f( int stat[] ) {  
 ^  
1 warning generated.  
Size of stat[23]: 92  
.. in function: 8

(This is an example of *pointer decay*)

You can dynamically reserve memory with `new`, which gives a something-star:

```
double *x;  
x = new double[27];
```

The `new` operator is only for C++: in C you would use `malloc` to dynamically allocate memory. The above example would become:

```
double *x;  
x = (double*) malloc( 27 * sizeof(double) );
```

Note that `new` takes the number of elements, and deduces the type (and therefore the number of bytes per element) from the context; `malloc` takes an argument that is the number of bytes. The `sizeof` operator then helps you in determining the number of bytes per element.

## 17.4 Pointer arithmetic

pointer arithmetic uses the size of the objects it points at:

```
double *addr_of_element = array;
cout << *addr_of_element;
addr_of_element = addr_of_element+1;
cout << *addr_of_element;
```

Increment add size of the array element, 4 or 8 bytes, not one!

**Exercise 17.2.** Write a subroutine that sets the i-th element of an array, but using pointer arithmetic: the routine should not contain any square brackets.

## 17.5 Multi-dimensional arrays

After

```
double x[10][20];
```

a row  $x[3]$  is a `double*`, so is  $x$  a `double**`?

Was it created as:

```
double **x = new double*[10];
for (int i=0; i<10; i++)
    x[i] = new double[20];
```

No: multi-d arrays are contiguous.

## 17.6 Parameter passing

C++ style functions that alter their arguments:

```
void inc(int &i) {
    i += 1;
}
int main() {
    int i=1;
    inc(i);
    cout << i << endl;
    return 0;
}
```

In C you can not pass-by-reference like this. Instead, you pass the address of the variable  $i$  by value:

```
void inc(int *i) {
    *i += 1;
}
int main() {
```

```

int i=1;
inc(&i);
cout << i << endl;
return 0;
}

```

Now the function gets an argument that is a memory address: *i* is an int-star. It then increases *\*i*, which is an int variable, by one.

Note how again there are two different uses of the ampersand character. While the compiler has no trouble distinguishing them, it is a little confusing to the programmer.

**Exercise 17.3.** Write another version of the *swap* function:

```

void swap( /* something with i and j */ {
    /* your code */
}

int main() {
    int i=1, j=2;
    swap( /* something with i and j */ );
    cout << "check that i is 2: " << i << endl;
    cout << "check that j is 1: " << j << endl;
    return 0;
}

```

Hint: write C++ code, then insert stars where needed.

### 17.6.1 Allocation

In section 10.10 you learned how to create arrays that are local to a scope:

Create an array with size depending on something:

```

if ( something ) {
    double ar[25];
} else {
    double ar[26];
}
ar[0] = // there is no array!

```

This Does Not Work

The array *ar* is created depending on if the condition is true, but after the conditional it disappears again. The mechanism of using *new* (section 17.6.2) allows you to allocate storage that transcends its scope:

C allocates in bytes:

```

double *array;
array = (double*) malloc( 25*sizeof(double) );

```

C++ allocates an array:

```

double *array;
array = new double[25];

```

## 17. C-style pointers and arrays

---

Don't forget:

```
free(array); // C  
delete array; // C++
```

Now dynamic allocation:

```
double *array;  
if (something) {  
    array = new double[25];  
} else {  
    array = new double[26];  
}
```

Don't forget:

```
delete array;
```

```
void func() {  
    double *array = new double[large_number];  
    // code that uses array  
}  
int main() {  
    func();  
};
```

- The function allocates memory
- After the function ends, there is no way to get at that memory
- ⇒ *memory leak*.

```
for (int i=0; i<large_num; i++) {  
    double *array = new double[1000];  
    // code that uses array  
}
```

Every iteration reserves memory, which is never released: another *memory leak*.

Your code will run out of memory!

Memory allocated with `malloc` / `new` does not disappear when you leave a scope. Therefore you have to delete the memory explicitly:

```
free(array);  
delete (array);
```

The C++ `vector` does not have this problem, because it obeys scope rules.

No need for `malloc` or `new`

- Use `std::string` for character arrays, and
- `std::vector` for everything else.

No performance hit if you don't dynamically alter the size.

### 17.6.1.1 Malloc

The keywords `new` and `delete` are in the spirit of C style programming, but don't exist in C. Instead, you use `malloc`, which creates a memory area with a size expressed in bytes. Use the function `sizeof` to translate from types to bytes:

```
int n;
double *array;
array = malloc( n*sizeof(double) );
if (!array)
    // allocation failed!
```

### 17.6.1.2 Allocation in a function

The mechanism of creating memory, and assigning it to a 'star' variable can be used to allocate data in a function and return it from the function.

```
void make_array( double **a, int n ) {
    *a = new double[n];
}
int main() {
    double *array;
    make_array(&array, 17);
}
```

Note that this requires a 'double-star' or 'star-star' argument:

- The variable `a` will contain an array, so it needs to be of type `double*`;
- but it needs to be passed by reference to the function, making the argument type `double**`;
- inside the function you then assign the new storage to the `double*` variable, which is `*a`.

Tricky, I know.

## 17.6.2 Use of `new`

*Before doing this section, make sure you study section 17.3.*

There is a dynamic allocation mechanism that is much inspired by memory management in C. Don't use this as your first choice.

Use of `new` uses the equivalence of array and reference.

```
// array/arraynew.cpp
void make_array( int **new_array, int length ) {
    *new_array = new int[length];
}
/* ... */
int *the_array;
make_array(&the_array, 10000);
```

Since this is not scoped, you have to free the memory yourself:

```
// array/arraynew.cpp
class with_array{
private:
    int *array;
    int array_length;
public:
    with_array(int size) {
        array_length = size;
        array = new int[size];
    };
    ~with_array() {
        delete array;
    };
}
/* ... */
with_array thing_with_array(12000);
```

Notice how you have to remember the array length yourself? This is all much easier by using a `std::vector`. See <http://www.cplusplus.com/articles/37Mf92yv/>.

The `new` mechanism is a cleaner variant of `malloc`, which was the dynamic allocation mechanism in C. Malloc is still available, but should not be used. There are even very few legitimate uses for `new`.

### 17.7 Memory leaks

Pointers can lead to a problem called a *memory leak*: there is memory that you have reserved, but you have lost the ability to access it.

In this example:

```
double *array = new double[100];
// ...
array = new double[105];
```

memory is allocated twice. The memory that was allocated first is never released, because in the intervening code another pointer to it may have been set. However, if that doesn't happen, the memory is both allocated, and unreachable. That's what memory leaks are about.

### 17.8 Const pointers

A pointer can be constant in two ways:

1. It points to a block of memory, and you can not change where it points.
2. What it points to is fixed, and the contents of that memory can also not be altered.

To illustrate the non-const behavior:

**Code:**

```
// pointer/starconst.cpp
int value = 5;
int *ptr = &value;
*ptr += 1;
cout << "value: " << value << '\n';
cout << "*ptr: " << *ptr << '\n';
ptr += 1;
cout << "random memory: " << *ptr << '\n';
```

**Output:**

```
[pointer] starconst1:
value: 6
*ptr: 6
random memory: 73896
```

A pointer that is constant in the first sense:

```
// pointer/starconst.cpp
int value = 5;
int *const ptr = &value;
*ptr += 1;
/* DOES NOT COMPILE: cannot assign to variable 'ptr' with const-qualified type 'int *const'
ptr += 1;
*/
```

You can also make a pointer to a constant integer:

```
// pointer/starconst.cpp
const int value = 5; // value is const
/* DOES NOT COMPILE: cannot convert const int* to int*
int *ptr = &value;
*/
```



# Chapter 18

## Const

The keyword `const` can be used to indicate that various quantities can not be changed. This is not an issue of program performance: it mostly serves programming safety: if you declare that a method will not change any members, and it does so (indirectly) anyway, the compiler will warn you about this.

Another way of looking at this is that using `const` expresses the intent of code: if something does not change, it should be marked as such. In fact, some recent programming languages have this `const` behavior as default, and things that can change have to be marked as such.

### 18.1 Const arguments

The use of `const` arguments is one way of protecting you against yourself. If an argument is conceptually supposed to stay constant, the compiler will catch it if you mistakenly try to change it.

Function arguments marked `const` can not be altered by the function code. The following segment gives a compilation error:

```
// const/constchange.cpp
void f(const int i) {
    ++i; // COMPILER ERROR!
}
```

### 18.2 Const references

A more sophisticated use of `const` is the `const reference`:

```
void f( const int &i ) { .... }
```

This may look strange. After all, references, and the pass-by-reference mechanism, were introduced in section 7.5 to return changed values to the calling environment. The `const` keyword negates that possibility of changing the parameter.

But there is a second reason for using references. Parameters are passed by value, which means that they are copied, and that includes big objects such as `std::vector`. Using a reference to pass a vector is much less costly in both time and space, but then there is the possibility of changes to the vector propagating back to the calling environment.

## 18. Const

---

Consider a class that has methods that return an internal member by reference, once as const reference and once not:

**Code:**

```
// const/constref.cpp
class has_int {
private:
    int mine{1};
public:
    const int& int_to_get() { return mine; }
    int& int_to_set() { return mine; }
    void inc() { ++mine; }
};

/* ...
has_int an_int;
an_int.inc(); an_int.inc();
printf("Contained int is now: {}", 
    an_int.int_to_get());
/* Compiler error: an_int.int_to_get() = 5;
*/
an_int.int_to_set() = 17;
printf("Contained int is now: {}", 
    an_int.int_to_get());
```

**Output:**  
[const] constref:  
Contained int is now: 4  
Contained int is now: 17

We can make visible the difference between pass by value and pass by const-reference if we define a class where the *copy constructor* explicitly reports itself:

```
// object/copyscalar.cpp
class has_int {
private:
    int mine{1};
public:
    has_int(int v) {
        cout << "set: " << v
        << '\n';
        mine = v; }
    has_int(has_int &h) {
        auto v = h.mine;
        cout << "copy: " << v
        << '\n';
        mine = v; }
    void printme() {
        cout << "I have: " << mine
        << '\n'; }
};
```

Now if we define two functions, with the two parameter passing mechanisms, we see that passing by value invokes the copy constructor, and passing by const reference does not:

**Code:**

```
// const/constcopy.cpp
void f_with_copy(has_int other) {
    cout << "function with copy"
    << '\n'; }
void f_with_ref(const has_int &other) {
    cout << "function with ref"
    << '\n'; }
/* ... */
cout << "Calling f with copy..." 
    << '\n';
f_with_copy(an_int);

cout << "Calling f with ref..." 
    << '\n';
f_with_ref(an_int);
```

**Output:**

[**const**] **constcopy**:

*Calling f with copy...  
(calling copy constructor)  
function with copy  
Calling f with ref...  
function with ref  
... done*

**18.2.1 Const references in range-based loops**

The same pass by value/reference issue comes up in range-based for loops. The syntax

```
for ( auto v : some_vector )
```

copies the vector elements to the *v* variable, whereas

```
for ( auto& v : some_vector )
```

makes a reference. To get the benefits of references (no copy cost) while avoiding the pitfalls (inadvertent changes), you can also use a const-reference here:

```
for ( const auto& v : some_vector )
```

**18.3 Const methods**

We can distinguish two types of methods: those that alter internal data members of the object, and those that don't. The ones that don't can be marked **const**. While this is in no way required, it contributes to a clean programming style:

Using **const** will catch mismatches between the declaration and definition of the method. For instance,

```
class Things {
    private:
    int var;
    public:
    f(int &ivar,int c) const {
        var += c; // typo: should be 'ivar'
    }
}
```

Here, the use of `var` was a typo, should have been `ivar`. Since the method is marked `const`, the compiler will generate an error because the function makes changes outside of its local scope.

It encourages a functional style, in the sense that it makes *side-effects* impossible:

```
class Things {
private:
    int i;
public:
    int getval() const { return i; }
    int inc() { return i++; } // side-effect!
    void addto(int &thru) const { // effect through parameter
        thru += i; }
```

Note that a method being `const` does not mean you can not call non-`const` functions in it: it means that the `this` object can not be affected.

## 18.4 Overloading on `const`

A `const` method and its non-`const` variant are different enough that you can use this for overloading and have them both.

Const and non-`const` version of `at`:

<b>Code:</b> <pre>// const/constat.cpp class has_array { private:     vector&lt;float&gt; values;; public:     has_array(int l, float v)         : values(vector&lt;float&gt;(l, v)) {}     auto&amp; at(int i) {         cout &lt;&lt; "var at" &lt;&lt; '\n';         return values.at(i); }     const auto&amp; at(int i) const {         cout &lt;&lt; "const at" &lt;&lt; '\n';         return values.at(i); }     auto sum() const {         float p;         for (int i=0; i&lt;values.size(); ++i)             p += at(i);         return p;     } };  int main() {      int l; float v;     cin &gt;&gt; l; cin &gt;&gt; v;     has_array fives(l, v);     cout &lt;&lt; fives.sum() &lt;&lt; '\n';     fives.at(0) = 2;     cout &lt;&lt; fives.sum() &lt;&lt; '\n'; }</pre>	<b>Output:</b> <pre>[const] constat: const at const at const at 1.5 var at const at const at const at 4.5</pre>
---	--

**Exercise 18.1.** Explore variations on this example, and see which ones work and which ones not.

1. Remove the second definition of `at`. Can you explain the error?
2. Remove either of the `const` keywords from the second `at` method. What errors do you get?

This mechanism can be made more elegant with the C++23 mechanism of *deducing this*; section 9.5.10.

## 18.5 Const and pointers

Let's declare a class `thing` to point to, and a class `has_thing` that contains a pointer to a `thing`.

```
// const/constpoint.cpp
class thing {
private:
    int i;
public:
    thing(int i) : i(i) {};
    void set_value(int ii) { i = ii; };
    auto value() const { return i; };
};

class has_thing {
private:
    shared_ptr<thing>
        thing_ptr=nullptr;
public:
    has_thing(int i)
        : thing_ptr
            (make_shared<thing>(i)) {};
    void print() const {
        cout << thing_ptr->value() << '\n';
    };
}
```

## 18. Const

---

If we define a method to return the pointer, we get a copy of the pointer, so redirecting that pointer has no effect on the container:

**Code:**

```
// const/constpoint.cpp
auto get_thing_ptr() const {
    return thing_ptr; }
/* ... */
has_thing container(5);
container.print();
container.get_thing_ptr() =
    make_shared<thing>(6);
container.print();
```

**Output:**  
[const] constpoint2:  
5  
5

If we return the pointer by reference we can change it. However, this requires the method not to be `const`. On the other hand, with the `const` method earlier we can change the object:

**Code:**

```
// const/constpoint.cpp
// Error: does not compile
// auto &get_thing_ptr() const {
auto &access_thing_ptr() {
    return thing_ptr; }
/* ... */
has_thing container(5);
container.print();
container.access_thing_ptr() =
    make_shared<thing>(7);
container.print();
container.get_thing_ptr()->set_value(8);
container.print();
```

**Output:**  
[const] constpoint3:  
5  
7  
8

If you want to prevent the pointed object from being changed, you can declare the pointer as a `shared_ptr<const thing>`:

```
// const/constpoint.cpp
private:
    shared_ptr<const thing>
        const_thing{nullptr};
public:
    has_thing(int i,int j)
        : const_thing
            (make_shared<thing>(i+j)) {};
    auto get_const_ptr() const {
        return const_thing; }
void crint() const {
    cout << const_thing->value() << '\n';
}
/* ... */
has_thing constainer(1,2);
// Error: does not compile
constainer.get_const_ptr()->set_value
(9);
```

### 18.5.1 Old-style const pointers

For completeness, a section on const and pointers in C.

We can have the `const` keyword in three places in the declaration of a C pointer:

```
int *p;
const int *p;
int const * p; // this is the same
int * const p; // as this
```

For the interpretation, it is often said to read the declaration ‘from right to left’. So:

```
int * const p;
```

is a ‘const pointer to int’. This means that it is const what int it points to, but that int itself can change:

```
// const/conststarconst.cpp
int i=5;
int * const ip = &i;
printf("ptr derefs to: %d\n", *ip);
*ip = 6;
printf("ptr derefs to: %d\n", *ip);
int j;
// DOES NOT COMPILE ip = &j;
```

On the other hand,

```
const int *p;
```

is a ‘pointer to a const int’. This means that you can point it at different ints, but you can not change the value those through the pointer:

```
// const/conststarconst.cpp
const int * jp = &i;
i = 7;
printf("ptr derefs to: %d\n", *jp);
// DOES NOT COMPILE *jp = 8;
int k = 9;
jp = &k;
printf("ptr derefs to: %d\n", *jp);
```

Finally,

```
const int * const p;
```

is a ‘const pointer to const int’. This pointer can not be retargeted, and the value of its target can not be changed:

```
// const/conststarconst.cpp
// DOES NOT WORK const int * const kp; kp = &k;
const int * const kp = &k;
printf("ptr derefs to: %d\n", *kp);
k = 10;
// DOES NOT COMPILE *kp = 11;
```

Because it can not be retargeted, you have to set its target when you declare such a pointer.

## 18.6    Mutable

Sometimes you may want methods that are marked `const`, but that still affect data outside of their scope. To illustrate this, consider a typical class with non-const update methods, and const readout of some computed quantity:

```
class Stuff {
private:
    int i, j;
public:
    Stuff(int i,int j) : i(i),j(j) {};
    void seti(int inew) { i = inew; };
    void setj(int jnew) { j = jnew; };
    int result () const { return i+j; };
```

Now suppose that the  $i+j$  computation in the `result` function stands for something expensive. You can solve that by

1. Caching the computed value;
2. The `result` function returns the cached value without further computation; and
3. You let the `seti`/`setj` methods recompute the cached value.

```
class Stuff {
private:
    int i, j;
    int cache;
    void compute_cache() { cache = i+j; };
public:
    Stuff(int i,int j) : i(i),j(j) { compute_cache(); };
    void seti(int inew) { i = inew; compute_cache(); };
    void setj(int jnew) { j = jnew; compute_cache(); };
    int result () const { return cache; };
```

For the next complication, assume that setting  $i, j$  happens far more often than requesting the computed value through the `result` method. The previous solution is then inefficient, and instead we:

1. Maintain a boolean flag that records whether the cached value is valid;
2. The `seti`/`j` function set this flag to `false`; and
3. The `result` function recomputes the cache, only if necessary.

```
class Stuff {
private:
    int i, j;
    int cache;
    bool cache_valid{false};
    void update_cache() {
        if (!cache_valid) {
            cache = i+j; cache_valid = true;
        }
    }
public:
    Stuff(int i,int j) : i(i),j(j) {};
    void seti(int inew) { i = inew; cache_valid = false; };
    void setj(int jnew) { j = jnew; cache_valid = false; }
    int result () const {
        update_cache(); return cache; };
```

This does not compile, because `result` is `const`, but it calls a non-`const` function.

We can solve this by declaring the cache variables to be `mutable`. Then the methods that conceptually don't change the object can still stay `const`, even while altering the state of the object. (It is better not to use `const_cast`.)

```

class Stuff {
private:
    int i, j;
    mutable int cache;
    mutable bool cache_valid{false};
    void update_cache() const {
        if (!cache_valid) {
            cache = i+j; cache_valid = true;
        }
    }
public:
    Stuff(int i,int j) : i(i),j(j) {};
    void seti(int inew) { i = inew; cache_valid = false; };
    void setj(int jnew) { j = jnew; cache_valid = false; };
    int result () const {
        update_cache(); return cache; };
}

```

## 18.7 Compile-time constants

Compilers have long been able to simplify expressions that only contain constants:

```

int i=5;
int j=i+4;
f(j)

```

Here the compiler will conclude that `j` is 9, and that's where the story stops. It also becomes possible to let `f(j)` be evaluated by the compiler, if the function `f` is simple enough. C++17 added several more variants of `constexpr` usage.

The `const` keyword can be used to indicate that a variable can not be changed:

```

const int i=5;
// DOES NOT COMPILE:
i += 2;

```

The combination `if constexpr` is useful with templates:

```

template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t;
    else
        return t;
}

```

To declare a function to be constant, use `constexpr`. The standard example is:

```

constexpr double pi() {
    return 4.0 * atan(1.0); };

```

but also

```

constexpr int factor(int n) {
    return n <= 1 ? 1 : (n*fact(n-1));
}

```

```
}
```

(Recursion in C++11, loops and local variables in C++14.)

- Can use conditionals, if testing on constant data;
- can use loops, if number of iterations constant;
- C++20 can allocate memory, if size constant.

## Chapter 19

### Declarations and header files

In this chapter you will learn techniques that you need for modular program design.

#### 19.1     Include files

You have seen the `#include` directive in the context of *header files* of the standard library, most notably the `iostream` header. But you can include arbitrary files, including your own.

To include files of your own, use a slightly different syntax:

```
#include "myfile.cpp"
```

(The angle bracket notation usually only works with files that are in certain system locations.) This statement acts as if the file is literally inserted at that location of the source.

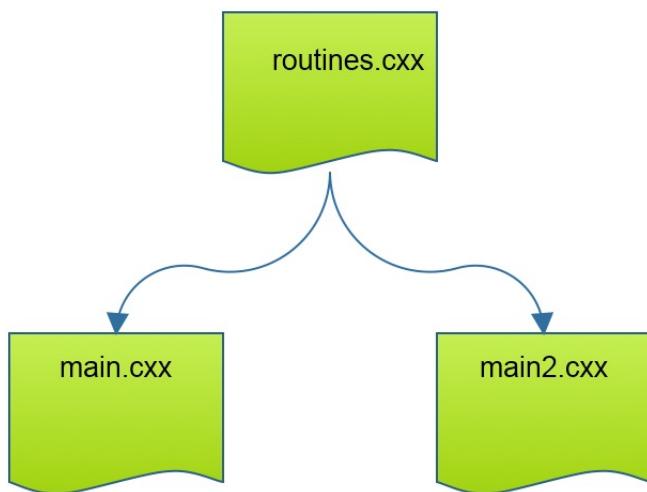


Figure 19.1: Using an include file for two main programs

This mechanism gives an easy solution to the problem of using some functions or classes in more than one main program; see figure 19.1.

The problem with this approach is that building the two main programs takes a time (roughly) equal to the sum of the compile times of the main programs and **twice** the compile time of the included file. Also, any time you change the included file you need to recompile the two main programs.

In a better scenario you would compile all three files once, and spend some little extra time tie-ing it all together. We will work towards this in a number of steps.

## 19.2 Function declarations

In most of the programs you have written in this course, you put any functions or classes above the main program, so that the compiler could inspect the definition before it encountered the use. However, the compiler does not actually need the whole definition, say of a function: it is enough to know its name, the types of the input parameters, and the return type.

Such a minimal specification of a function is known as *function prototype*; for instance

```
int tester(float);
```

Some people like defining functions after the main.

Problem: the main needs to know about them.

'forward declaration'

```
int f(int);
int main() {
    f(5);
}
int f(int i) {
    return i;
}
```

versus:

```
int f(int i) {
    return i;
}
int main() {
    f(5);
}
```

This is a stylistic choice.

You also need forward declaration for mutually recursive functions:

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

Declarations are useful if you spread your program over multiple files. You would put your functions in one file and the main program in another.

```
// file: def.cpp
int tester(float x) {
    ....
}

// file : main.cpp
int main() {
    int t = tester(...);
    return 0;
}
```

In this example a function `tester` is defined in a different file from the main program, so we need to tell `main` what the function looks like in order for the main program to be compilable:

```
// file : main.cpp
int tester(float);
int main() {
    int t = tester(...);
    return 0;
}
```

Splitting your code over multiple files and using *separate compilation* is good software engineering practice for a number of reasons.

1. If your code gets large, compiling only the necessary files cuts down on compilation time.
2. Your functions may be useful in other projects, by yourself or others, so you can reuse the code without cutting and pasting it between projects.
3. It makes it easier to search through a file without being distracted by unrelated bits of code.

(However, you would not do things like this in practice. See section 19.2.2 about header files.)

### 19.2.1 Separate compilation

Your regular compile line

```
icpc -o yourprogram yourfile.cpp
```

actually does two things: compilation, and linking. You can do those separately:

1. First you compile

```
icpc -c yourfile.cpp
```

which gives you a file `yourfile.o`, a so-called *object file*; and

2. Then you use the compiler as *linker* to give you the *executable file*:

```
icpc -o yourprogram yourfile.o
```

In this particular example you may wonder what the big deal is. That will become clear if you have multiple source files: now you invoke the compile line for each source, and you link them only once.

Compile each file separately, then link:

```
icpc -c mainfile.cc
icpc -c functionfile.cc
icpc -o yourprogram mainfile.o functionfile.o
```

At this point, you should learn about the *Make* tool for managing your programming project.

### 19.2.2 Header files

Even better than writing the declaration every time you need the function is to have a *header file*:

Using a header file with function declarations.

Header file contains only declaration:

```
// file: def.h
int tester(float);
```

The header file gets included twice:

Definitions file:

```
// compile/testerdef.cpp
#include "def.h"
int tester(float x) {
    ....
}
```

Main program:

```
// compile/testermain.cpp
#include "def.h"
int main() {
    int t = tester(...);
    return 0;
}
```

What happens in both cases if you leave out the `#include "def.h"`?

Having a header file is an important safety measure: it allows the compiler to check that use and definition of a class or function are compatible.

It is necessary to include the header file in the file that uses the function or class. It is not strictly necessary to include it in the definitions file, but doing so means that you catch potential errors, as argued by C++ Core Guidelines [8], SF.5.

**Remark** By the way, why does that compiler even recompile the main program, even though it was not changed? Well, that's because you used a makefile. See Tutorials book [11], section 3.

**Remark** Header files were able to catch more errors in C than they do in C++. With polymorphism of functions, it is no longer an error to have

```
// header.h
int somefunction(int);
```

and

```
#include "header.h"

int somefunction( float x ) { .... }
```

### 19.2.3 C and C++ headers

You have seen the following syntaxes for including header files:

```
#include <header.h>
#include "header.h"
```

The first is typically used for system files, with the second typically for files in your own project. There are some header files that come from the C standard library such as `math.h`; the idiomatic way of including them in C++ is

```
#include <cmath>
```

### 19.3 Declarations for class methods

Section 9.5.3 explained how you can split a class declaration from its definition. You can then put the declaration in a header file that you include in every file where is a class is used, while the definitions get compiled only once, and linked in when the executable of your program is built, or put in a library.

Header file:

```
// proto/header.hpp
class something {
private:
    int i;
public:
    double dosomething( int i, char c );
};
```

Implementation file:

```
// proto/func.cpp
double something::dosomething( int i, char c ) {
    // do something with i,c
};
```

There are two things to remark on here:

1. The header contains not only the class name, and function member names, but also names of data members, including the private ones!
2. Function member definitions have their name prefixed with the class name and a double colon. Also, they repeat qualifiers such as `const`.

**Review 19.1.** For each of the following answer: is this a valid function definition or function declaration.

- `int foo();`
- `int foo() {};`
- `int foo(int) {};`
- `int foo(int bar) {};`
- `int foo(int) { return 0; };`
- `int foo(int bar) { return 0; };`

### 19.4 More

#### 19.4.1 Header files and templates

See section 22.2.3.

#### 19.4.2 Namespaces and header files

Namespaces (see chapter 20) are convenient, but they carry a danger in that they may define functions without the user of the namespace being aware of it.

Therefore, one should never put `using namespace` in a header file.

### 19.4.3 Global variables and header files

If you have a variable that you want known everywhere, you can make it a *global variable*:

```
int processnumber;
void f() {
    ... processnumber ...
}
int main() {
    processnumber = // some system call
};
```

It is then defined in the main program and any functions defined in your program file.

Warning: it is tempting to define variables global but this is a dangerous practice; see section [20.4](#).

If your program has multiple files, you should not put ‘`int processnumber`’ in the other files, because that would create a new variable, that is only known to the functions in that file. Instead use:

```
extern int processnumber;
```

which says that the global variable `processnumber` is defined in some other file.

What happens if you put that variable in a *header file*? Since the *preprocessor* acts as if the header is textually inserted, this again leads to a separate global variable per file. The solution then is more complicated:

```
//file: header.h
#ifndef HEADER_H
#define HEADER_H
#ifndef EXTERN
#define EXTERN extern
#endif
EXTERN int processnumber
#endif

//file: aux.cc
#include "header.h"

//file: main.cc
#define EXTERN
#include "header.h"
```

(This sort of preprocessor magic is discussed in chapter [21](#).)

This also prevents recursive inclusion of header files.

## 19.5 Modules

The C++20 standard is taking a different approach to header files, through the introduction of *modules*. (Fortran90 already had this for a long time; see chapter [37](#).) This largely dispenses with the need for header files included through the C Preprocessor (CPP). However, the CPP may still be needed for other purposes.

### 19.5.1 Program structure with modules

Using modules, the `#include` directive is no longer needed; instead, the `import` keyword indicates what module is to be used in a (sub)program:

```
// modulex/fgmain.cpp
import fg_module;
int main() {
    std::cout << "Hello world " << f(5) << '\n';
```

The module is usually in a different file. The module name does not derive from that file name; instead the `export` keyword, followed by `module` defines the name of the module. This file can then have any number of functions; only the ones with the `export` keyword will be visible in a program that imports the module.

```
// modulex/fgmod.cpp
export module fg_module;

// internal function
int g( int i ) { return i/2; };

// exported function
export int f( int i ) {
    return g(i+1);
};
```

### 19.5.2 Implementation and interface units

#### 19.5.3 Partitions

A module can have a leveled structure, by using names with a `modulename:partition` structure.

This makes it possible to have separate

- Interface partitions, that define the interface to the using program; and
- Implementation partitions, that contain the code that needs to be kept from the user.

Here is an implementation partition; there is no `import` keyword because this functionality is internal:

```
// modulex/helperhelp.cpp
// implementation unit, nothing exported
module helper_module:helper;
// internal function
int g( int i ) { return i/2; };
```

Here is a file with an interface partition, which uses the internal function, and an exported function:

```
// modulex/helpermod.cpp
export module helper_module;
import :helper;

// exported function
export int f( int i ) {
    return g(i+1);
};
```

#### 19.5.4 More

Importable headers:

```
import <header.h>
import "header.h"
```

Import declarations have to come before other module specifications, whether import or export of modules or functions.

You can export variables, namespaces.

# Chapter 20

## Namespaces

### 20.1 Solving name conflicts

You have probably many times used the `vector` class that implements dynamic arrays. But what if you are writing a geometry package, containing a class for geometric vectors that you would like to call ‘vector’? Is there confusion with the `vector` class of the standard library? There would be if it weren’t for the phenomenon *namespace*, which acts as a disambiguating prefix for classes, functions, variables.

In a way you have already seen namespaces in action. After including the `vector` header, you write

```
std::vector<int> bunch_of_ints;
```

which gives you the `vector` class from the `std` namespace. Thus, the double colon separates a namespace and an identifier, whether that be a class, function, or variable name.

You can make your own namespaces, with the `namespace` keyword.

Introduce new namespace:

```
namespace geometry {
    // definitions
    class vector {
        // stuff
    };
}
```

To access a class defined in a namespace, you qualify the class name with the namespace.

Double-colon notation for namespace and type:

```
geometry::vector myobject();
```

or

```
using geometry::vector;
vector myobject();
```

or even

```
using namespace geometry;
vector myobject();
```

Some packages have a complicated namespace structure, so you may find yourself writing

```
namespace abc = space_a::space_b::space_c;
abc::func(x)
```

In the C++ standard library, having two levels is quite common.

## 20.2 Namespace header files

If your namespace is going to be used in more than one program, you want to have it in a separate source file, with an accompanying header file.

Suppose we have a *geometry* namespace containing a *vector*, in addition to the *vector* in the standard namespace.

```
// namespace/geo.cpp
#include <vector>
#include "geolib.hpp" // this contains the geometry namespace
int main() {
    // std vector of geom segments:
    std::vector<geometry::segment> segments;
    segments.push_back( geometry::segment( geometry::point(1,1), geometry::point(4,5) ) );
}
```

What would the implementation of this be?

The header file would contain the function and class headers, but now inside a named namespace:

```
// namespace/geolib.hpp
namespace geometry {
    class point {
        private:
            double xcoord, ycoord;
        public:
            point() {};
            point( double x, double y );
            double x();
            double y();
    };
    class vector {
        private:
            point from,to;
        public:
            vector( point from, point to );
            double size();
    };
}
```

and the implementation file would have the implementations, in a namespace of the same name:

```
// namespace/geolib.cpp
namespace geometry {
    point::point( double x, double y ) {
        xcoord = x; ycoord = y; }
    double point::dx( point other ) {
        return other.xcoord - xcoord; }
    /* ... */

    template< typename T >
    vector<T>::vector( std::string name, int size )
        : name_(name), std::vector<T>::vector(size) {}
}
```

**Exercise 20.1.** Add a `vector` class to the `geometry` namespace, which inherits from `std::vector`, and which makes the following code work:

```
// namespace/geo.cpp
std::vector<geometry::vector<float>> vectors;
vectors.push_back( geometry::vector<float>( "a", 5 ) );
cout << fmt::format("First vector, \"{}\" has size {}\\n",
                    vectors[0].name(), vectors[0].size());
```

## 20.3 Namespace versioning for libraries

As the introduction to this topic argued, namespaces are a good way to prevent name conflicts. This means that it's a good idea to create a namespace for all your routines. You see this design in almost all published C++ libraries.

Now consider this scenario:

1. You write a program that uses a certain library;
2. A new version of the library is released and installed on your system;
3. Your program, using shared/dynamic libraries, starts using this library, maybe even without you realizing it.

This means that the old and new libraries need to be compatible in several ways:

1. All the classes, functions, and data structures defined in the earlier version also need to be defined in the new. This is not a big problem: new library versions typically add functionality. However,
2. The data layout of the new version needs to be the same.

That second point is subtle. To illustrate, consider the library is upgraded:

First version:

```
namespace geometry {
    class vector {
        private:
            std::vector<float> data;
            std::string name;
    }
}
```

New version:

```
namespace geometry {
    class vector {
        private:
            std::vector<float> data;
            int id;
            std::string name;
    }
}
```

The problem is that your compiled program has explicit information where the class members are located in the class object. For instance, in the first version above, the `name` member may come 64 bytes from the start of the object; in the second version it will then be at 68 bytes from the start of the object.

By changing the structure of the objects, such byte offsets are no longer correct. This is called ‘Application Binary Interface (ABI) breakage’ and it leads to *undefined behavior*.

The library can prevent this by:

```
namespace geometry {
    inline namespace v1.0 {
        class vector {
            private:
                std::vector<float> data;
                std::string name;
        }
    }
}
```

and updating the version number in future version. A program using this library has to use explicitly the namespace version: `geometry::v1.0::vector`.

Example:

Namespace definition with versions:

```
// namespace/inline.cpp
namespace geometry {
    inline namespace v1 {
        class vector {
            public: float x, y;
        };
    }
    inline namespace v2 {
        class vector {
            public: float x, y, z;
        };
    }
}
```

Versioned use:

```
// namespace/inline.cpp
{
    using geometry::v1::vector;
    vector p; p.x = 1.f;
}
{
    using geometry::v2::vector;
    vector p; p.z = 1.f;
}
```

## 20.4 Best practices

In this course we advocated indicating the namespace of functions with a `using` statement:

```
#include <iostream>
using std::cout;
```

(The alternative being to write `std::cout` everywhere.)

It is also possible to use

```
#include <iostream>
using namespace std;
```

The problem with this is that it may make available unwanted functions. In the following example it leads to an incorrect code being compiled, and possibly executing with an erroneous result.

Illustrating the dangers of `using namespace std`:

This compiles, but should not:

```
// func/swapname.cpp
#include <iostream>
using namespace std;

def swap(int i,int j) {};

int main() {
    int i=1,j=2;
    swap(i,j);
    cout << i << '\n';
    return 0;
}
```

This gives an error:

```
// func/swapusing.cpp
#include <iostream>
using std::cout;

def swap(int i,int j) {};

int main() {
    int i=1,j=2;
    swap(i,j);
    cout << i << '\n';
    return 0;
}
```

Even if you write `using namespace`, you should only do this in a source file, never in a header file. Any-one using the header would have no idea what functions are suddenly defined; C++ Core Guidelines [8], SF.7.



# Chapter 21

## Preprocessor

In your source files you have seen lines starting with a hash sign, such as

```
#include <iostream>
```

Lines starting with a hash character are called *directives*, and they are interpreted by the C preprocessor, also simply called the *preprocessor*. This is a source-to-source translation stage that comes before the actual compiler.

We will see some of the more common uses of the preprocessor here.

Best practice note.

The preprocessor is capable of powerful effects, that are in fact not achievable otherwise. However, since it also leads to dangerous and confusing programming practices, its use should be kept minimal.

### 21.1 Include files

The `#include` directive causes the named file to be included. That file is in fact an ordinary text file, either part of your own sources, part of some library you are using, or a header that is stored somewhere in your system. As a result, your source file is transformed to another source file in a source-to-source translation stage, and only that second file is actually compiled by the *compiler*.

Normally, this intermediate file with all include files textually included is immediately destroyed again after the compilation, but in rare cases you may want to dump it for debugging. See your compiler documentation for how to generate this file.

Compiler	Commandline	Notes
gcc	<code>-fpreprocessed</code>	
Intel	<code>-E</code>	writes to standard out

Compiling the hello-world program, the C preprocessor generates a text file where all preprocessor directives have been resolved, for instance by textual inclusion of include files. In the case of the hello world program in C, that's the `stdlib.h` and `stdio.h` files. The preprocessor output file will look something like:

```
# 0 "hello.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "hello.c"
# 12 "hello.c"
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX13.sdk/usr/include/stdlib.h" 1 3 4
# 61 "/Library/Developer/CommandLineTools/SDKs/MacOSX13.sdk/usr/include/stdlib.h" 3 4
# 1 "/Library/Developer/CommandLineTools/SDKs/MacOSX13.sdk/usr/include/Availability.h" 1 3 4
```

et cetera.

### 21.1.1 Kinds of includes

While you can include any kind of text file, the most common use of the preprocessor is to include a *header file* at the start of your source using a CPP `#include` directive; section 21.1.

There are two kinds of includes

1. The file name can be included in angle brackets,

```
#include <vector>
```

which is typically used for system headers that are part of the compiler infrastructure;

2. The name can also be in double quotes,

```
#include "somedata.h"
```

which is typically used for header files that are part of your code, or of libraries installed on your system.

### 21.1.2 Search paths

System headers can usually be found by the compiler because they are in some standard location. Including other files may require additional action by you. If you write

```
#include "foo.h"
```

the compiler only looks for that file in the current directory where you are compiling. So the question is how to tell the compiler to look for this file elsewhere.

If the include file is part of a library that you have downloaded and installed yourself, say it is in

```
/home/yourname/mylibrary/include/foo.h
```

then you could of course spell out the location:

```
#include "/home/yourname/mylibrary/include/foo.h"
```

However, that does not make your code very portable to other systems and other users. So how do you make

```
#include "foo.h"
```

be understood on any system?

The answer is that you can give your compiler one or more *include paths*. This is done with the `-I` flag.

```
icpc -c yourprogram.cpp -I/home/yourname/mylibrary/include
```

You can specify more than one such flag, and the compiler will try to find the `foo.h` file in all of them.

Are you now thinking that you have to type that path every time you compile? This is the time to learn about the utilities `Make` (Tutorials book [11], chapter 3) or `CMake` (Tutorials book [11], chapter 4).

**Remark** Some libraries put headers in a subdirectory of the `include` directory. Your source would then include the header with the last level of the `include` path:

```
// Eigen library:  
#include "Eigen/Core"  
// fmtlib  
#include <fmt/format.h>
```

This mechanism is used to prevent name collisions, such as having two header files named `format.h`.

## 21.2 Textual substitution

Suppose your program has a number of arrays and loop bounds that are all identical. To make sure the same number is used, you can create a variable, and pass that to routines as necessary.

```
void dosomething(int n) {  
    for (int i=0; i<n; i++) ....  
}  
int main() {  
    int n=100000;  
    vector<int> idxs(n);  
    vector<float> vals(n);  
    dosomething(n);
```

You can also use `#define` to define a *preprocessor macro*:

```
#define N 100000  
void dosomething() {  
    for (int i=0; i<N; i++) ....  
}  
int main() {  
    vector<int> idxs(N);  
    vector<float> vals(N);  
    dosomething();
```

It is traditional to use all uppercase for such macros.

### 21.2.1 Dynamic definition of macros

Your code may occasionally have compile-time constants, such as the size of a `std::array`. If you want to change this value, do you have to edit and recompile your program? You can skip the edit stage by using a macro for the value:

```
std::array<int,N> fixed_array;
```

and have the value of `N` be defined on the compile line, using the `-D` flag:

```
icpc -c yourprogram.cpp -DN=100000
```

Now what if you want a default value in your source, but optionally redefine it in the compilation stage? You can solve this by testing for definition in the source with `#ifndef`: ‘if not defined’:

```
#ifndef N
#define N 10000
#endif
```

See section [21.3.2](#) for more on ‘define’ directives.

### 21.2.2 Parametrized macros

Instead of simple text substitution, you can have *parametrized preprocessor macros*. As an example, the following macro checks for return codes that require a routine to exit prematurely:

```
#define CHECK_FOR_ERROR(i) if (i!=0) return i
...
ierr = some_function(a,b,c); CHECK_FOR_ERROR(ierr);
ierr = more_function(d,e);   CHECK_FOR_ERROR(ierr);
```

You see that such parametrized macros look like a little like function definitions, except that

- the parameters have no type: they are substituted textually;
- the macro substitution is not a scope, unless you explicitly surround the expansion text in curly braces;
- the whole definition has to fit on one line; you can escape the line end with a backslash if you are writing a long macro.

When you introduce parameters, it’s a good idea to use lots of parentheses. The following definition is dangerous:

```
#define MULTIPLY(a,b) a*b
...
x = MULTIPLY(1+2, 3+4);
```

This expands to

```
1+2 * 3+4
```

A better version uses parentheses:

```
#define MULTIPLY(a,b) ( (a)*(b) )
...
x = MULTIPLY(1+2, 3+4);
```

Another popular use of macros is to simulate multi-dimensional indexing:

```
#define INDEX2D(i,j,n) (i)*(n)+j
...
double array[m*n];
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    array[ INDEX2D(i, j, n) ] = ...
```

**Exercise 21.1.** Write a macro that simulates 1-based indexing:

```
#define INDEX2D1BASED(i,j,n) /* your definition here */
...
double array[m*n];
for (int i=1; i<=m; i++)
    for (int j=n; j<=n; j++)
        array[ INDEX2D1BASED(i, j, n) ] = ...
```

### 21.2.3 Type definitions

Related to macros is the `typedef` keyword:

```
typedef int* intptr;
intptr my_ptr;
```

The `using` keyword that you saw in section 4.2.1 can also be used as a replacement for `typedef` if it's used to introduce synonyms for types.

```
using Matrix = vector<vector<float>>;
```

## 21.3 Conditionals

There are two types of *preprocessor conditionals*: test for defined macros, and tests on a numeric expression. We start with a common idiom.

### 21.3.1 Disable a block of code

The `#if` macro tests on the nonzero value of its following expression. A common application is to temporarily remove code from compilation:

```
#if 0
    bunch of code that needs to
    be disabled
#endif
```

### 21.3.2 Check for defined macros

The `#ifdef` test tests for a macro being defined. Conversely, `#ifndef` tests for a macro not being defined. For instance,

```
#ifndef N
#define N 100
#endif
```

Why would a macro already be defined? Well, you can do that on the compile line:

```
icpc -c file.cc -DN=500
```

The C++23 standard introduced additionally `#elifdef` and `#elifndef`.

### 21.3.3 Numeric expression

You can also test on C-like expressions. The `#if 0` idiom was actually a special case of this, since zero is interpreted as false.

Here are some examples in which macros are implicitly expanded:

```
#if VARIANT == 1
    some code
#elif CUDA_VERSION > 12000
    other code
#else
#error Invalid combination
#endif
```

Note the `#error` pragma, which is a good idea if you have many nested conditions.

Another application for this test is in preventing recursive inclusion of header files; see section [19.4.3](#).

### 21.3.4 Including a file only once

It is easy to wind up including a file such as `iostream` more than once if it is included in multiple header files. This adds to your compilation time, or may lead to subtle problems. A header file may even circularly include itself. To prevent this, header files often have what is informally known as a *header guard*.

Header file tests if it has already been included:

```
// this is foo.h
#ifndef FOO_H
#define FOO_H

// the things that you want to include

#endif
```

This prevents double or recursive inclusion.

Now the file will effectively be included only once: the first time it is processed in full and the `FOO_H` macro is defined; the second time the `#ifndef` test fails and the file content after the test is skipped.

Many compilers support the pragma `#pragma once` that has the same effect:

```
// this is foo.h
#pragma once

// the things you want to include only once
```

However, this is not standardized, and the precise meaning is unclear (what if this is placed halfway a file?) so C++ Core Guidelines [8], SF.8 recommends the explicit guards.

## 21.4 Other preprocessor directives

Packing data structure without padding bytes by `#pack`

```
#pragma pack(push, 1)
// data structures
#pragma pack(pop)
```

If you have too many `#ifdef` cases, you may get combinations that don't work. In order to exit compilations that don't make sense, use the `#error` program:

```
#ifdef __vax__
#error "Won't work on VAXen."
#endif
```

Less serious is `#warning` which will only put a warning string on your screen.

In section 21.3.4 you saw the `#pragma` directive. This has some standardized uses, but it can also be used for compiler-specific specifications.



## Chapter 22

### Templates

You have seen how objects of type `vector<string>` and `vector<float>` are very similar: both have methods with the same names, and these methods behave largely the same. This angle-bracket notation is called ‘templating’ and the `string` or `float` is called the *template parameter*.

If you go digging into the source of the C++ library, you will find that, somewhere, there is just a single definition of the `vector` class, but with a new notation it gets the type `string` or `float` as template parameter.

It is as if (and reality is of course more complicated) the templated class (or function) is preceded by a line

```
template< typename T >
class vector {
    ...
};
```

This mechanism is available for your own functions and classes too.

**Remark** Historically `typename` was `class` but that’s confusing. However, you may still come across this in older code.

#### 22.1 Templatized functions

We will start by taking a brief look at templated functions.

For this example observe that  $\sqrt{3.14} = 1.77200451467$ .

Definition:

```
// template/func.cpp
template <typename Real>
void sqrt_diff( Real x ) {
    cout << std::sqrt(x)-1.772 << '\n';
}
```

We use this with a templated function:

<b>Code:</b>	<b>Output:</b>
<pre>// template/func.cpp sqrt_diff&lt;float&gt; ( 3.14f ); sqrt_diff&lt;double&gt; ( 3.14 );</pre>	<pre>[template] func: 4.48513e-06 4.51467e-06</pre>

**Exercise 22.1.** Machine precision, or ‘machine epsilon’, is sometimes defined as the smallest number  $\epsilon$  so that  $1 + \epsilon > 1$  in computer arithmetic.

Write a templated function `machine_eps` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

<b>Code:</b>	<b>Output:</b>
<pre>// template/eps.cpp float float_eps =     machine_eps&lt;float&gt;(); cout &lt;&lt; "Epsilon float: "     &lt;&lt; setw(10) &lt;&lt; setprecision(4)     &lt;&lt; float_eps &lt;&lt; '\n';  double double_eps =     machine_eps&lt;double&gt;(); cout &lt;&lt; "Epsilon double: "     &lt;&lt; setw(10) &lt;&lt; setprecision(4)     &lt;&lt; double_eps &lt;&lt; '\n';</pre>	<pre>[template] eps: Epsilon float: 1.1921e-07 Epsilon double: 2.2204e-16</pre>

Hint: you may need to cast scalars to the appropriate type.

**Exercise 22.2.** If you’re doing the zero-finding project, you can now do the exercises in section 47.2.3.

## 22.2 Templatized classes

The most common use of templates is probably to define templated classes. For instance, you could templatize the `Point` class of section 46.2:

Coordinates can be <code>float</code> or <code>double</code> :
<pre>// geom/pointtemplate.cpp template&lt;typename T&gt; class Point { private:     T x, y; public:     Point(T ux, T uy) { x = ux; y = uy; };</pre>

Coordinates can also be other things, but that doesn’t always make sense.

You have in fact seen this mechanism in action with the `std::vector` class, which is templated over the type of element contained in the vector.

The templated vector class looks roughly like:

```
template<typename T>
class vector {
private:
    T *vectordata; // internal data
public:
    T at(int i) { return vectordata[i]; }
    int size() { /* return size of data */ }
    // much more
}
```

Let's consider a worked out example. We write a class `Store` that stored a single element of the type of the template parameter.

Intended behavior:

**Code:**

```
// template/example1.cpp
Store<int> i5(5);
cout << i5.stored_value() << '\n';
```

**Output:**

```
[template] example1i5:
```

5

The class definition is preceded by a `template` line, and uses the template parameter throughout.

Template parameter is used for private data, return type, etc.

```
// template/example1.cpp
template< typename T >
class Store {
private:
    T stored;
public:
    Store(T v) : stored(v) {};
    T stored_value() { return stored; };
```

**Exercise 22.3.** Take your `Point` class from a previous exercise and templatize the class definition.

Write the `distance` function for the templated class Write a main program that tests this.

If we write methods that refer to the templated type, things get a little more complicated. Let's say we want two methods `copy` and `negative` that return objects of the same templated type:

The method definitions are fairly straightforward; if you leave out the template parameter, the *class name injection* mechanism uses the same template value as for the class being defined. In the following example the template parameter can be specified or left out, and the **CTAD!** mechanism will deduce the missing template parameter.

### 22.2.1 Out-of-class method definitions

If we separate the class declaration and the method definitions (for instance for separate compilation; see section 19.3) things get trickier. The class declaration is basically the class definition minus method bodies.

Declaration of a templated class:

```
// template/example2.cpp
template< typename T >
class Store {
private:
    T stored;
public:
    Store(T v);
    T value() const;
    Store copy() const;
    Store<T> negative() const;
};
```

The method definitions are more tricky. Now the template parameter needs to be specified every single time you mention the templated class, except for the name of the constructor:

Each method needs the `template` line:

```
// template/example2.cpp
template< typename T >
Store<T>::Store(T v) : stored(v) {};

template< typename T >
T Store<T>::value() const { return stored; };

template< typename T >
Store<T> Store<T>::copy() const { return Store<T>(stored); };

template< typename T >
Store<T> Store<T>::negative() const { return Store<T>(-stored); };
```

If your declaration and definitions are in separate files, there is a further complication; see section [22.2.3](#).

### 22.2.2 Specific implementations: template specializations

Sometimes the template code works for a number of types (or values), but not for all. In that case you can specify the instantiation for specific types with empty angle brackets:

```
template <typename T>
void f(T);
template <>
void f(char c) { /* code with c */ };
template <>
void f(double d) { /* code with d */ };
```

### 22.2.3 Templates and separate compilation

The use of templates often makes *separate compilation* hard: in order to compile the templated definitions the compiler needs to know with what types they will be used. For this reason, many libraries are set up as *header-only library*: they have to be included in each file where they are used, rather than compiled separately and linked.

In the case where you can foresee which types a templated class will be instantiated with, there is a way to have separate compilation of the library routines. Suppose you have a templated class and it will only be used (instantiated) with certain types:

Class definition:

MISSING SNIPPET instant in  
codesnippetsdir=snippets.code

Only instantiations:

```
// namespace/instant.cpp
instant<char> ic;
ic.out();
instant<int> ii;
ii.out();
```

We can then add, for each type, a line to the implementation file, stating that the class will be instantiated with that type.

## 22.3 Example: polynomials over fields

It makes sense for numerical applications to be templated to allow for computation in *single precision floats*, *double precision doubles*, and perhaps `std::complex` numbers. However, we can often also generalize the computation to other *fields*. Consider as an example polynomials, in both scalars and (square) matrices.

Let's start with a simple class for polynomials:

```
// polynomial/poly_eval.cpp
class polynomial {
private:
    vector<double> coefficients;
public:
    polynomial( vector<double> c )
        : coefficients(c) {};
    // 5 x^2 + 4 x + 3 = 5 x + 4 x + 3
    double eval( double x ) const {
        double y{0.};
        for_each(coefficients.rbegin(), coefficients.rend(),
            [x,&y] (double c) { y *= x; y += c; } );
        return y;
    };
    double operator()(double x) const { return eval(x); };
}
```

We store the polynomial coefficients, with the zeroth-degree coefficient in location zero, et cetera. The routine for evaluating a polynomial for a given input  $x$  is an implementation of *Horner's rule*:

$$5x^2 + 4x + 3 \equiv ((5) \cdot x + 4) \cdot x + 3$$

(Note that the `eval` method above uses `rbegin`, `rend` to traverse the coefficients in the correct order from last to first.)

For instance, the coefficients  $2, 0, 1$  correspond in increasing order of  $x$  to the polynomial  $2 + 0 \cdot x + 1 \cdot x^2$ :

```
// polynomial/poly_eval.cpp
polynomial x2p2( {2., 0., 1.} );
for ( auto x : {1., 2., 3.} ) {
    auto y = x2p2(x);
    cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}
```

We can generalize the above to the case of matrices, where both the  $x$  input and  $y$  output are matrices. For now we let the polynomial coefficients be scalars.

The above code for evaluating a polynomial for a certain input works just as well for matrices, as long as the multiplication and addition operator are defined. So let's say we have a class `mat` and we have member functions:

```
// polynomial/poly_mat.cpp
mat operator+( const mat& other ) const;
void operator+=( const mat& other );
mat operator*( const mat& other ) const;
mat operator*( double other ) const;
void operator*=( const mat& other );
void operator*=( double other );
```

Now we redefine the `polynomial` class, templated over the scalar type:

```
// polynomial/poly_mat.cpp
template< typename Scalar >
class polynomial {
private:
    vector<double> coefficients;
public:
    polynomial( vector<double> c )
        : coefficients(c) {};
    int degree() const { return coefficients.size()-1; }
    // 5 x^2 + 4 x + 3 = 5 x + 4 x + 3
    Scalar eval( Scalar x ) const {
        Scalar y{0.};
        for_each(coefficients.rbegin(),coefficients.rend(),
                 [x,&y] (double c) { y *= x; y += c; } );
        return y;
    };
};
```

The code using polynomials stays the same, except that we have to supply the scalar type as template parameter whenever we create a polynomial object. The above example of  $p(x) = x^2 + 2$  becomes for scalars:

```
// polynomial/poly_mat.cpp
polynomial<double> x2p2( {2., 0., 1.} );
for ( auto x : {1., 2., 3.} ) {
    auto y = x2p2(x);
    cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}
```

and for matrices:

```
// polynomial/poly_mat.cpp
polynomial<mat> x2p2( {2., 0., 1.} );
```

```

for ( auto x : {1., 2., 3.} ) {
    auto y = x2p2(x);
    cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
}

```

**Exercise 22.4.** Can you extend the matrix class to allow for matrix-valued coefficients?

## 22.4 Concepts

The C++20 standard added the notion of *concept*.

Templates can be too generic. For instance, one could write a templated *gcd* function

```

template <typename T>
T gcd( T a, T b ) {
    if (b==0) return a;
    else return gcd(b, a%b);
}

```

which will work for various integral types. To prevent it being applied to non-integral types, you can specify a *concept* to the type:

```

// template/concept.cpp
#include <type_traits>
template<typename T>
concept Integral = std::is_integral_v<T>;

```

used as:

```

// template/concept.cpp
template <typename T>
requires Integral<T>
T gcd0( T a, T b ) {
    if (b==0) return a;
    else return gcd0(b, a%b);
}

```

or

```

// template/concept.cpp
template <Integral T>
T gcd1( T a, T b ) {
    if (b==0) return a;
    else return gcd1(b, a%b);
}

```

We can even dispense with the `template` keyword altogether:

```

// template/concept.cpp
Integral auto gcd
    ( Integral auto a, Integral auto b ) {
    if (b==0) return a;
    else return gcd(b, a%b);
}

```

## 22. Templates

---

Now we use this:

**Code:**

```
// template/concept.cpp
int i=27, j=30;
cout << "gcd(27,30)="
    << gcd(i,j) << '\n';
long ii=27'000'000'000,
      jj=30'000'000'000;
cout << "gcd(27G,30G)="
    << gcd(ii,jj) << '\n';
// WRONG: DOES NOT COMPILE
// gcd(27.1,30.2);
```

**Output:**

[**template**] concept:

gcd(27,30)=3  
gcd(27G,30G)=3000000000

## Chapter 23

### Error handling

When you're programming, making errors is close to inevitable. Violations of the grammar of the language, known as *syntax errors*, will be caught by the compiler. These will prevent generation of an executable. In this section we will talk about *runtime errors*: behavior at runtime that is other than intended.

Here are some sources of runtime errors

**Array indexing** Using an index outside the array bounds may give a runtime error:

```
vector<float> a(10);
for (int i=0; i<=10; i++)
    a.at(i) = x; // runtime error
```

or undefined behavior:

```
vector<float> a(10);
for (int i=0; i<=10; i++)
    a[i] = x;
```

**Null pointers** Using an uninitialized pointer is likely to crash your program:

```
Object *x;
if (false) x = new Object;
x->method();
```

**Numerical errors** such as division by zero will not crash your program, so catching them takes some care.

A productive way of dealing with errors is a combination of preventing them and handling them when they occur. A block-structured code is then valuable: you can approach code block as

```
{
    // start of block: some preconditions hold
    // code of the block: deal with errors
    // end of block: some post-conditions hold
}
```

Checking pre and post-conditions with assertions will be discussed in section 23.1; dealing with errors through exceptions is the topic of section 23.2.

### 23.1 Assertions

One way catch errors before they happen, is to sprinkle your code with assertions: statements about things that have to be true from the logic of your program. For instance, if a function should mathematically always return a positive result, such as a valid square root of a scalar, it may be a good idea to check for that anyway. You do this by using the `assert` command, which takes a boolean, and will stop your program if the boolean is false:

Check on valid input parameters:

```
#include <cassert>

// this function requires x<y
// it computes something positive
float f(x, y) {
    assert( x<y );
    return /* some computation */;
}
```

Code design to facilitate testing the result:

```
float positive_outcome = /* some computation */;
assert( positive_outcome>0 );
return positive_outcome;
```

The `static_assert` statement checks compile-time conditions only.

Since checking an assertion is a minor computation, you may want to disable it when you run your program in production by defining the `NDEBUG` macro:

```
#define NDEBUG
```

One way of doing that is passing it as a compiler flag:

```
icpc -DNDEBUG yourprog.cpp
```

As an example of using assertions, we can consider the iteration function of the Collatz exercise 6.14.

```
// collatz/modular.cpp
int collatz_next( int current ) {
    assert( current>0 );
    int next{-1};
    if (current%2==0) {
        next = current/2;
        assert(next<current);
    } else {
        next = 3*current+1;
        assert(next>current);
    }
    return next;
}
```

**Remark** If an assertion fails, your program will call `std::abort`.

## 23.2    Exception handling

During the run of your program, an error condition may occur that requires you to take one or more steps back to deal with it. For cases where these errors are exceptional, typically caused by unforeseeable runtime conditions, we have the *exception* mechanism.

You may have already seen exception in action: if you access a vector element outside the bounds of that vector, your program will stop. You may see text on your screen

*terminating with uncaught exception*

We say that your program has *thrown* an exception.

**Code:**

```
// except/boundthrow.cpp
vector<float> x(5);
x.at(5) = 3.14;
```

**Output:**

```
[except] boundthrow:
libc++abi.dylib: terminating
→with uncaught exception
→of type
→std::out_of_range:
→vector
```

Now you know that your program has there is an error, but you don't know where it occurs. You can find out, by trying to *catch the exception*.

**Code:**

```
// except/catchbounds.cpp
vector<float> x(5);
for (int i=0; i<10; ++i) {
    try {
        x.at(i) = 3.14;
    } catch (...) {
        cout << "Exception indexing at: "
            << i << '\n';
        break;
    }
}
```

**Output:**

```
[except] catchbounds:
Exception indexing at: 5
```

The operative terms here are

- *exception throwing*, which uses the `throw` keyword to signal a problem; and
- *exception catching*, which uses the `catch` keyword to specify actions to be taken when an exception is thrown.

Illustration of exception throwing and catching.

Code with problem:

```
if ( /* some problem */ )
    throw(5);
```

Calling code:

```
try {
    /* code that can go wrong */
} catch (...) { // literally three dots!
    /* code to deal with the problem */
}
```

You can throw all sorts of exceptions; in the example just now we used an integer, but you can throw an error string, or even exception objects. More below.

### 23.2.1 Standard exceptions

Exceptions that the standard library throws, such as for `at` indexing out of bounds, or allocation an invalid vector size, inherit from `std::exception`. In particular they have a `what` method that returns an informative string:

**Code:**

```
// except/what.cpp
case 1 :
    [] (int s) { vector<int> x(s); }(-3);
    break;
//
case 2 :
    [] (int s) { vector<int> x(3); x.at(s); }(4);
    break;
//
case 3 :
    [] () { throw(5); }();
    break;
```

**Output:**

```
[except] what:
For experiment 1 found
    ↪standard exception:
    cannot create std::vector
        ↪larger than max_size()
For experiment 2 found
    ↪standard exception:
    vector::M_range_check:
        ↪n (which is 4) >=
        ↪this->size() (which is
        ↪3)
For experiment 3 found
    ↪unknown exception
```

### 23.2.2 Popular exceptions

- `std::out_of_range`: usually caused by using `at` with an invalid index.

### 23.2.3 Exception types

In the examples above it didn't matter much what you threw, since the `catch` clause only detected that there was an exception, period. You can be much more sophisticated. For instance, you could throw an integer corresponding to some error code.

The `catch` clause now explicitly indicates that it's expecting an integer as the content of the exception:

If you're doing the prime numbers project, you can now do exercise 45.12.

You can actually throw any type of object as an exception. For instance, you can throw integers to indicate an error code, or a string with an actual error message. You could even make an error class:

```

class MyError {
public :
    int error_no; string error_msg;
    MyError( int i, string msg )
        : error_no(i), error_msg(msg) {};
}

throw( MyError(27, "oops") ;

try {
    // something
} catch ( MyError &m ) {
    cout << "My error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
}

```

You can use exception inheritance!

If different levels of your function call hierarchy throw different types of exceptions, you could have multiple **catch** clauses to deal with the different types.

You can multiple **catch** statements to catch different types of errors:

```

try {
    // something
} catch ( int i ) {
    // handle int exception
} catch ( std::string c ) {
    // handle string exception
}

```

You can catch any exception regardless the type by specifying three dots. This can also be used as a final case in a sequence of **catch** clauses.

Catch exceptions without specifying the type:

```

try {
    // something
} catch ( ... ) { // literally: three dots
    cout << "Something went wrong!" << endl;
}

```

### Exercise 23.1. Define the function

$$f(x) = x^3 - 19x^2 + 79x + 100$$

and evaluate  $\sqrt{f(i)}$  for the integers  $i = 0 \dots 20$ .

- First write the program naively, and print out the root. Where is  $f(i)$  negative? What does your program print?
- You see that floating point errors such as the root of a negative number do not make your program crash or something like that. Alter your program to throw an exception if  $f(i)$  is

negative, catch the exception, and print an error message.

- Alter your program to test the output of the `sqrt` call, rather than its input. Use the function `isnan`

```
#include <cfenv>
using std::isnan;
```

and again throw an exception.

In a somewhat rare case, imagine that you have a constructor that could throw an exception, and this constructor is used as a base case for a derived class:

```
class A{
public:
    A(int i) { /* ... */
        if (something) throw(something);
    };
};

class B : public A {
public:
    B(int i)
        : A(i) { /* something */ };
};
```

How do you now catch the exception thrown in the `A` constructor?

A function `try` block will catch exceptions, including in *member initializer* lists of constructors.

```
B::B( int i )
try : A(i) {
    // constructor body
}
catch (...) { // handle exception
}
```

Some remarks.

- Functions were able to define what exceptions they throw until C++17:
 

```
void func() throw( MyError, std::string );
```
- There are many predefined exceptions: `bad_alloc`, `bad_exception`, etc.
- An exception handler can throw an exception; to rethrow the same exception use ‘`throw;`’ without arguments.
- Exceptions delete all stack data, but not `new` data. Also,
- The main program acts as if an implicit `try/except` block is placed around it. You can replace the handler for that. See the `exception` header file.
- The keyword `noexcept` indicates that a function can not throw an exception:
 

```
void f() noexcept { ... };
```

 This can be important in some applications such as embedded systems.
- No exception is thrown when dereferencing a `nullptr`.

### 23.3 Remaining topics

#### 23.3.1 ‘Where does this error come from’

The CPP defines two macros, `__FILE__` and `__LINE__` that give you respectively the current file name and the current line number. You can use these to generate pretty error messages such as

*Overflow occurred in line 25 of file numerics.cpp*

The C++20 standard offers the `source_location` header as a mechanism instead.

```
// except/location.cpp
#ifndef _CPP_LIB_SOURCE_LOCATION
    std::source_location loc_info
        = std::source_location::current();
    cout << "Exception at line " << loc_info.line << '\n';
#else
    cout << "Exception at unknown source location\n";
#endif
```

### 23.3.2 Legacy mechanisms

The traditional approach to error checking is for each routine to return an integer parameter that indicates success or absence thereof. Problems with this approach arise if it's used inconsistently, for instance by a user forgetting to heed the return codes of a library. Also, it requires that every level of the function calling hierarchy needs to check return codes.

The PETSc library uses this mechanism consistently throughout, and to great effect.

Exceptions are a better mechanism, since

- they can not be ignored, and
- they do not require handling on the levels of the calling hierarchy between where the exception is thrown and where it is caught.

Also, throw exceptions will automatically de-allocate the memory from standard containers.

As an alternative to exceptions, consider `optional` (section 24.6.2) or `expected` (section 24.6.3).



## Chapter 24

### Standard Template Library

The C++ language has a Standard Library (formerly known as *Standard Template Library (STL)*), which contains functionality that is considered standard, but that is actually implemented in terms of already existing language mechanisms. The STL is enormous, so we just highlight a couple of parts.

You have already seen

- arrays (chapter 10),
- strings (chapter 11),
- streams (chapter 12).

Using a template class typically involves

```
#include <something>
using std::function;
```

see section 20.1.

#### 24.1 Complex numbers

Complex numbers require the `complex` header. The `complex` type uses templating to set the precision, and the real/imaginary part of a number are available through the `.re` and `.im` members.

```
#include <complex>
std::complex<float> f;
f.re = 1.; f.im = 2.;

complex<double> d(1., 3.);
```

Math operators such as `+`, `*` are defined between complex numbers. Numerical expressions involving a complex number and a simple scalar are well-defined if the scalar is of the underlying type of the complex number:

```
complex<float> x;
x + 1.f; // Yes
x + 1.; // No, because '1.' is double
```

You can use the imaginary unit number  $i$  through the literals `i`, `if` (float), `il` (long):

```
using namespace std::complex_literals;
std::complex<double> c = 1.0 + 1i;
```

Beware: `1+1i` does not compile: you need to write `1.+1i` for the compiler to deduce the types.

Operations on vector of complex:

**Code:**

```
// complex/veccomplex.cpp
vector<complex<double>>
vec1( N, 1.+2.5i );
auto vec2( vec1 );
/* ... */
for ( int i=0; i<vec1.size(); ++i ) {
    vec2[i] = vec1[i] * ( 1.+1.i );
}
/* ... */
auto sum = accumulate
( vec2.begin(), vec2.end(),
  complex<double>(0.) );
println( "result: {}", sum );
```

**Output:**

```
[complex] vec:
result: (-1.5e+06,3.5e+06)
```

There are support functions for such things as exponentiation (`exp`) and complex conjugation (`conj`), as well as `abs`, `norm`, `polar`.

### 24.1.1 Complex support in C

The C language has had complex number support since C99 with the types

```
float _Complex
double _Complex
long double _Complex
```

The header `complex.h` gives synonyms

```
float complex
double complex
long double complex
```

for these.

See for instance <https://en.cppreference.com/w/c/numeric/complex> for details.

## 24.2 Limits

Numeric limits can be queried with the `std::numeric_limits` function of the `limit` header. Inherited from C, there is a header file `limits.h` / `climits` containing macros such as `MAX_INT` and `MIN_INT`, but this should not be used.

Use header file `limits`:

```
#include <limits>
using std::numeric_limits;
```

Limits for signed and unsigned integers:

**Code:**

```
// int/unsigned.cpp
#include <iostream>
#include <limits>

int main() {
    std::cout << "max int      : {" << std::numeric_limits<int>::max() << "}" << std::endl;
    std::cout << "min int      : {" << std::numeric_limits<int>::min() << "}" << std::endl;
    std::cout << "max unsigned  : {" << std::numeric_limits<unsigned int>::max() << "}" << std::endl;
    std::cout << "min unsigned  : {" << std::numeric_limits<unsigned int>::min() << "}" << std::endl;
}
```

**Output:**

[int] limit:

```
max int      : 2147483647
max unsigned : 4294967295
```

For floating point numbers more limit quantities are defined.

- The largest number is given by `max`;
- `min` is the smallest positive number that is not a denormal; use `lowest` for ‘most negative’.
- The smallest denormal number is given by `denorm_min`.
- There is an `epsilon` function for machine precision:

Limits of floating point numbers:

**Code:**

```
// stl/limits.cpp
#include <iostream>
#include <limits>

int main() {
    std::cout << "Single lowest {" << std::numeric_limits<float>::lowest() << "}" << std::endl;
    std::cout << "... and epsilon {" << std::numeric_limits<float>::epsilon() << "}" << std::endl;
    std::cout << "Double lowest {" << std::numeric_limits<double>::lowest() << "}" << std::endl;
    std::cout << "... and epsilon {" << std::numeric_limits<double>::epsilon() << "}" << std::endl;
}
```

**Output:**

[stl] eps:

```
Single lowest -3.40282e+38 and
→epsilon 1.19209e-07
Double lowest -1.79769e+308 and
→epsilon 2.22045e-16
```

**Exercise 24.1.** Write a program to discover what the maximal  $n$  is so that  $n!$ , that is,  $n$ -factorial, can be represented in an `int`, `long`, or `long long`. Can you write this as a templated function?

Operations such as dividing by zero lead to floating point numbers that do not have a valid value. For efficiency of computation, the processor will compute with these as if they are any other floating point number.

## 24.3 Containers

C++ has several types of *containers*. You have already seen `std::vector` (section 10.3) and `std::array` (section 10.4) and strings (chapter 11). Many containers have methods such as `push_back` and `insert` in common.

In this section we will look at a couple more containers.

### 24.3.1 Maps: associative arrays

Arrays use an integer-valued index. Sometimes you may wish to use an index that is not ordered, or for which the ordering is not relevant. A common example is looking up information by string, such as finding the age of a person, given their name. This is sometimes called ‘indexing by content’, and the data structure that supports this is formally known as an **associative array**.

In C++ associative arrays are implemented through a `map`:

```
#include <map>
using std::map;
map<string, int> ages;
```

is set of pairs where the first item (which is used for indexing) is of type `string`, and the second item (which is found) is of type `int`.

A map is made by inserting the elements one-by-one:

```
#include <map>
using std::make_pair;
ages.insert(make_pair("Alice", 29));
ages["Bob"] = 32;
```

You can range over a map:

```
for ( auto person : ages )
    println("{} has age {}", person.first, person.second);
```

A more elegant solution uses *structured binding* (section 24.5):

```
for ( auto [person, age] : ages )
    println("{} has age {}", person, age);
```

(It is possible to use const-references here.)

Searching for a key gives either the iterator of the key/value pair, or the `end` iterator if not found:

```
// map/countint.cpp
for ( auto k : {4,5} ) {
    auto wherek = intcount.find(k);
    if (wherek==intcount.end())
        println("could not find key {}", k);
    else {
        auto [kk, vk] = *wherek;
        println("found key: {} has value {}", kk, vk);
    }
}
```

**Remark** Iterating over a `map` returns results sorted by key. There is a `unordered_map` that can have speed and storage advantages but is not sorted.

**Exercise 24.2.** If you’re doing the prime number project, you can now do the exercises in section 45.6.3.

### 24.3.2 Sets

The `set` container is like a `map<SomeType, bool>`, that is, it only says ‘this element is present’. Like a mathematical set, in other words.

```
#include <set>
std::set<int> my_ints;
my_ints.insert(5);
my_ints.empty(); // predicate
my_ints.size(); // obvious
```

Range iteration is defined, giving the elements in no particular order:

```
for ( auto x : my_ints )
    // do something
```

You can search through a set with `find`, which results in an iterator. If no match is found the `end` iterator is returned.

```
auto itr = my_ints.find(6);
if ( itr==my_ints.end() )
    println("not found");
```

You can search on elements that satisfy a predicate with `find_if`:

```
auto res = find_if(my_ints.begin(),my_ints.end(),
    [] ( auto e ) { return e>37; } );
```

Sets are useful in many computer science application such as graph algorithms; see for instance HPC book [13], section 10.1.1. You often encounter idioms such as

```
while (not done) {
    for ( x in unprocessed ) {
        if (something with x) {
            processed.add(x);
            unprocessed.remove(x);
        }
    }
}
```

## 24.4 Regular expression

The header `regex` gives C++ the functionality for *regular expression* matching. For instance, `regex_match` returns whether or not a string matches an expression exactly:

**Code:**

```
// regexp/regexp.cpp
auto cap = regex("[A-Z] [a-z]+");
for ( auto n :
    { "Victor", "aDam", "DoD" }
) {
    auto match =
        regex_match( n, cap );
    cout << n;
    if (match) cout << ": yes";
    else cout << ": no" ;
    cout << '\n';
}
```

**Output:**

```
[regexp] regexp:
Looks like a name:
Victor: yes
aDam: no
DoD: no
```

(Note that the regex matches substrings, but `regex_match` only returns true for a match on the whole string.

For finding substrings, use `regex_search`:

- the function itself evaluates to a `bool`;
- there is an optional return parameter of type `smatch` ('string match') with information about the match.

The `smatch` object has these methods:

- `smatch::position` states where the expression was matched,
- while `smatch::str` returns the string that was matched.
- `smatch::prefix` has the string preceding the match; with `smatch::prefix().size()` you get the number of characters preceding the match, that is, the location of the match.

<b>Code:</b> <pre>// regexp/regsearch.cpp {     auto findthe = regex("the");     auto found = regex_search         ( sentence, findthe );     assert( found==true );     cout &lt;&lt; "Found &lt;&lt;the&gt;&gt;" &lt;&lt; '\n'; } {     smatch match;     auto findthx = regex("o[^o]+o");     auto found = regex_search         ( sentence, match , findthx );     assert( found==true );     cout &lt;&lt; "Found &lt;&lt;o[^o]+o&gt;&gt;"          &lt;&lt; " at " &lt;&lt; match.position(0)         &lt;&lt; " as &lt;&lt;" &lt;&lt; match.str(0) &lt;&lt; "&gt;&gt;"         &lt;&lt; " preceeded by &lt;&lt;" &lt;&lt; match.prefix () &lt;&lt; "&gt;&gt;"          &lt;&lt; '\n'; }</pre>	<b>Output:</b> <b>[regexp] search:</b> <pre>Found &lt;&lt;the&gt;&gt; Found &lt;&lt;o[^o]+o&gt;&gt; at 12 as     ↗&lt;&lt;own fo&gt;&gt; preceeded by     ↗&lt;&lt;The quick br&gt;&gt;</pre>
--	---

#### 24.4.1 Regular expression syntax

C++ uses a variant of the International regular expression syntax. <http://ecma-international.org/ecma-262/5.1/#sec-15.10>. Consult that document for escape characters and more.

If your regular expression is getting too complicated with escape characters and such, consider using the *raw string literal* construct.

## 24.5 Tuples and structured binding

A tuple is a way to pack together variables, possibly of different types. An object of the class `tuple` is parametrized by its constituent types:

```
tuple<int, char, double> icd;
```

Such an object is created with the `make_tuple` function:

```
auto icd = make_tuple(5, 'd', 3.14);
```

Note that the function is templated, but usually it's easier to leave out the types and have them deduced.

Conversely, a tuple can be unpacked into its components, using a method called *structured binding*:

```
auto [i, c, d] = icd;
// or by reference:
auto& [i, c, d] = icd;
```

Unpacking by reference is a little tricky: the variables `i`, `c`, `d` behave largely as if they are of type `int`, `char`, `double`, but they are actually components in a reference to the tuple. You may notice this if you pass them as function argument.

For pairs, there is the class `pair` with the function `make_pair`. This is equivalent to a tuple with two elements, except that there are now data members `first` and `second`.

### 24.5.1 Example: roots of a quadratic equation

As an illustration of the above, we consider computing the roots  $x_+, x_-$  of the quadratic equation  $ax^2 + bx + c = 0$ . We could define a `quadratic` class, but it suffices to use a `using` statement:

```
// union/abctuple.cpp
using quadratic = tuple<double, double, double>;
/* ... */
// polynomial: x^2 - 2
auto sunk = quadratic(1., 0., -2);
```

You can now use structured bindings to unpack the tuple:

#### Discriminant of the quadratic polynomial

Definition:

```
// union/abctuple.cpp
double discriminant
( quadratic q ) {
    auto [a,b,c] = q;
    return b*b-4*a*c;
}
```

Use:

```
// union/abctuple.cpp
auto d = discriminant( sunk );
cout << "discriminant: "
     << d << '\n';
```

Now use structured bindings to return the two roots:

**Exercise 24.3.** Write the function `abc_roots` that makes this code work:

```
// union/abctuple.cpp
auto roots = abc_roots( sunk );
auto [xplus, xminus] = roots;
cout << xplus << "," << xminus << '\n';
```

### 24.5.2 Example: square root

Remember how in section 7.5.2 we said that if you wanted to return more than one value, you could not do that through a return value, and had to use an *output parameter*? Well, using the standard library there is a different solution.

You can make a *tuple* with `tuple`: an entity that comprises several components, possibly of different type, and which unlike a `struct` you do not need to define beforehand. (For tuples with exactly two elements, use `pair`.)

In C++11 you would use the `get` method to extract elements from a pair or tuple.

Use `get` to unpack a tuple:

```
#include <tuple>

std::tuple<int,double,char> id = \
    std::make_tuple<int,double,char>( 3, 5.12, 'f' );
// or:
std::make_tuple( 3, 5.12, 'f' );
double result = std::get<1>(id);
std::get<0>(id) += 1;

// also:
std::pair<int,char> ic = make_pair( 24, 'd' );
```

Annoyance: all that ‘get’ting.

This does not look terribly elegant. Fortunately, C++17 can use denotations and the `auto` keyword to make this considerably shorter.

Consider the case of a function that returns a tuple. You could use `auto` to deduce the return type:

```
// stl/tuple.cpp
auto maybe_root1(float x) {
    if (x<0)
        return make_tuple
            <bool,float>(false,-1);
    else
        return make_tuple
            <bool,float>
                (true,sqrt(x));
}
```

but more interestingly, you can use a *tuple denotation*:

```
// stl/tuple.cpp
tuple<bool,float>
maybe_root2(float x) {
    if (x<0)
        return {false,-1};
    else
        return {true,sqrt(x)};
```

Here the return type of the function is indicated, but the `return` statements do not construct this type explicitly.

The calling code is particularly elegant:

<b>Code:</b> <pre>// stl/tuple.cpp auto [succeed,y] = maybe_root2(x); if (succeed)     cout &lt;&lt; "Root of " &lt;&lt; x     &lt;&lt; " is " &lt;&lt; y &lt;&lt; '\n'; else     cout &lt;&lt; "Sorry, " &lt;&lt; x     &lt;&lt; " is negative" &lt;&lt; '\n';</pre>	<b>Output:</b> <pre>[stl] tuple: Root of 2 is 1.41421 Sorry, -2 is negative</pre>
--	--

This is known as *structured binding*.

The above examples, with one component of the tuple being a `bool`, are better formulated differently, as you'll see below. An more appropriate use of structured binding is iterating over a map (section 24.3.1):

```
for ( const auto &[key,value] : mymap ) ....
```

## 24.6 Union-like stuff: optionals, variants, expected

There are cases where you need a value that is one type or another, for instance, a number if a computation succeeded, and an error indicator if not.

Let us consider the example of a square root function that returns an error if you call it with a negative number. The simplest solution is to have a function that returns both a bool and a number:

Function:

```
// union/optroot.cpp
bool RootOrError(float &x) {
    if (x<0)
        return false;
    else
        x = std::sqrt(x);
    return true;
};
```

Use:

```
// union/optroot.cpp
for ( auto x : {2.f,-2.f} )
    if (RootOrError(x))
        cout << "Root is "
        << x << '\n';
    else
        cout
            << "could not take root of "
            << x << '\n';
```

There are various inelegancies with this solution. For one, you're using two different mechanisms for returning the two values. Furthermore, you don't actually need two values: if there is an error, the returned value is irrelevant.

We will now consider some more idiomatically C++17 solutions to this.

### 24.6.1 Tuples

Using tuples or pairs (section 24.5) the solution to the above ‘a number or an error’ now becomes a tuple of a `bool` and a `float`:

```
// union/optroot.cpp
#include <tuple>
using std::tuple, std::pair;
```

```

/* ... */
pair<bool, float> RootAndValid(float x) {
    if (x<0)
        return {false,x};
    else
        return {true, std::sqrt(x)};
}
/* ... */
for ( auto x : {2.f,-2.f} )
    if ( auto [ok,root] = RootAndValid(x) ; ok )
        cout << "Root is "
        << root << '\n';
    else
        cout
            << "could not take root of "
            << x << '\n';

```

Note the *initializer statement* in the conditional.

This solution is better in that the bool and the result are now returned together through the function result, rather than through a parameter by reference, but we still have the problem of redundancy.

## 24.6.2 Optional

The most elegant solution to ‘a number or an error’ is to have a single quantity that you can query whether it’s valid. For this, the C++17 standard introduced the concept of a *nullable type*: a type that can somehow convey that it’s empty.

In this section we discuss `std::optional` which uses the `optional` header:

```
#include <optional>
using std::optional;
```

### 24.6.2.1 Creating an optional

The function we are writing has a return type of `optional<float>`. This is realized by returning either an actual `float`, or `{ }`, which is a synonym for `std::nullopt`.

```
optional<float> f {
    if (something)
        // result if success
        return 3.14;
    else
        // indicate failure
        return {};
}
```

### 24.6.2.2 Getting the optional value, if any

You can test whether the optional quantity actually has a quantity with the method `has_value`, in which case you can extract the quantity with the method `value`. Alternatively, you can use the fact that the `optional` object can be implicitly cast to `bool`, and dereferenced:

Two ways:

```
// union/optroot.cpp
for ( auto x : {2.f,-2.f} )
    if ( auto root = MaybeRoot(x);
        root.has_value() )
        cout << "Root is "
        << root.value()
        << '\n';
    else
        cout
        << "could not take root of "
        << x << '\n';
```

```
// union/optroot.cpp
for ( auto x : {2.f,-2.f} )
    if ( auto root = MaybeRoot(x) ; root )
        cout << "Root is "
        << *root << '\n';
    else
        cout
        << "could not take root of "
        << x << '\n';
```

Trying to take the `value` for something that doesn't have one leads to a `bad_optional_access` exception:

**Code:**

```
// union/optional.cpp
optional<float> maybe_number = {};
try {
    cout << maybe_number.value() << '\n';
} catch (std::bad_optional_access) {
    cout << "failed to get value\n";
}
```

**Output:**

```
[union] optional:
failed to get value
```

There is a function `value_or` that gives the value, or a default if the optional did not have a value.

```
optional<int> some_int;
some_int.value_or(-1);
```

**Exercise 24.4.** Continue the ‘abc’ exercise above and write a function that returns optionally a string,

```
// union/abctuple.cpp
optional<string> how_many_roots( quadratic q );
```

saying ‘one’ or ‘two’ if there are real roots:

```
// union/abctuple.cpp
auto num_solutions = how_many_roots(sunk);
if ( not num_solutions.has_value() )
    cout << "none\n";
else
    cout << num_solutions.value() << '\n';
```

**Exercise 24.5.** Write the `RootOrError` function using `std::optional`.

**Exercise 24.6.** The *eight queens* problem (chapter 48) can be elegantly solved using `std::optional`. See also section 48.3 for a Test-Driven Development (TDD) approach.

**Exercise 24.7.** If you are doing the prime number project, you can now do exercise 45.18.

**Remark** If you have an optional class object, you can assign that object with the `emplace` method:

Class definition:

```
// union/optional.cpp
class WithInt {
public:
    WithInt( int i ) {};
    void foo() {};
};
```

Emplacing into optional:

```
// union/optional.cpp
optional<WithInt> maybe;
{ maybe.emplace(5); }
cout << maybe.has_value() << '\n';
maybe.value().foo();
```

### 24.6.2.3 Monadic extensions

In C++23, `std::optional` and `std::expected` received *monadic extensions*: ways of further processing the optional result, but only if it has a value.

Code:

```
// union/optroot.cpp
optional<float> MaybeRoot(float x);
for ( auto x : {2.f,-2.f} )
    MaybeRoot(x)
        .transform( [] ( float x ) { return x*x;
        } )
        .and_then( [] (float x ) -> optional<
float> {
            cout << "check root is: "
            << x << '\n'; return x; } );
```

Output:

```
[union] monroot:
check root is: 2
```

### 24.6.3 Expected

The `std::optional` of the previous section is great for cases such as the square root, where it is clear what it means if the value does not exist. However, if the non-existence comes from some sort of error, you may want to ask what the reason for it is.

The C++23 addition of `std::expected` (in the `expected` header) allows you to return a value, or provide more information on why that error is not there, with `std::unexpected`.

Expect double, return info string if not:

Function returning `expected`:

```
// union/expected.cpp
std::expected<double, string>
square_root( double x ) {
    if (x<0)
        return std::unexpected("negative");
    else if (x<limits<double>::min())
        return std::unexpected("underflow");
    else {
        auto result = sqrt(x);
        return result;
    }
}
```

In Use:

```
// union/expected.cpp
auto root = square_root(x);
if (x)
    cout << "Root=" << root.value() << '\n';
else if (root.error() == /* et cetera */)
    /* handle the problem */
```

#### 24.6.4 Variant

In C/C++, a `union` is an entity that can be one of a number of types. Similar to how C arrays do not know their size, a `union` does not know what type it is. The C++ `variant` (C++17) does not suffer from these limitations.

In the above `optional` solution we have a `value` function. Since a `variant` can have multiple types, we can not directly have such a function. Instead, we need to test on what type is being returned, and use `get` or `get_if` to retrieve a value by type.

It is safer to use `get_if` which takes a pointer to a variant, and returns a pointer if successful, and a `null pointer` if not:

```
// union/intdoublestring.cpp
union_ids = "Hello world";
if ( auto union_int = get_if<int>(&union_ids) ; union_int )
    cout << "Int: " << *union_int << '\n';
else if ( auto union_string = get_if<string>(&union_ids) ; union_string )
    cout << "String: " << *union_string << '\n';
```

For a first example we consider the square root example that we started this chapter with, although this example is actually better done with an `optional`.

Return square root or false result:

```
// union/optroot.cpp
#include <variant>
using std::variant, std::get_if;
/* ... */
variant<bool, float> RootVariant(float x)
{
    if (x<0)
        return false;
    else
        return std::sqrt(x);
};
```

```
// union/optroot.cpp
for ( auto x : {2.f, -2.f} ) {
    auto okroot = RootVariant(x);
    auto root = get_if<float>(&okroot);
    if ( root )
        cout << "Root is "
        << *root << '\n';
    auto nope = get_if<bool>(&okroot);
    if ( nope )
        cout
            << "could not take root of "
            << x << '\n';
}
```

If we have more than two possibilities, `get_if` may get cumbersome. With `index` it is possible to query directly which of the variants is used.

Illustrating the usage:

```
// union/intdoublestring.cpp
variant<int,double,string> union_ids;
```

The `index` returns 0, 1, 2, and we `get` the value accordingly:

```
// union/intdoublestring.cpp
union_ids = 3.5;
switch ( union_ids.index() ) {
case 1 :
    cout << "Double case: " << std::get<double>(union_ids) << '\n';
}
```

Getting the wrong variant leads to a `bad_variant_access` exception:

```
// union/intdoublestring.cpp
union_ids = 17;
cout << "Using option " << union_ids.index()
     << ":" << get<int>(union_ids) << '\n';
try {
    cout << "Get as double: " << get<double>(union_ids) << '\n';
} catch ( bad_variant_access b ) {
    cout << "Exception getting as double while index="
        << union_ids.index() << '\n';
}
```

**Exercise 24.8.** Write a routine that computes the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

The routine should return two roots, or one root, or an indication that the equation has no solutions.

```
// union/abctuple.cpp
auto root_cases = abc_cases( sunk );
switch (root_cases.index()) {
case 0 : cout << "No roots\n"; break;
case 1 : cout << "Single root: " << get<1>(root_cases); break;
case 2 : {
    auto xs = get<2>(root_cases);
    auto [xp,xm] = xs;
    cout << "Roots: " << xp << ", " << xm << '\n';
} ; break;
}
```

**Exercise 24.9.** Write a routine that computes the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

The routine should return two roots, or one root, or an indication that the equation has no solutions.

```
Code:
// union/quadratic.cpp
for ( auto coefficients :
    { quadratic{.a=2.0, .b=1.5, .c=2.5},
      quadratic{.a=1.0, .b=4.0, .c=4.0},
      quadratic{.a=2.2, .b=5.1, .c=2.5}
    } ) {
  auto result = compute_roots(coefficients);
```

```
Output:
[union] quadratic:
With a=2 b=1.5 c=2.5
No root
With a=2.2 b=5.1 c=2.5
Root1: -0.703978 root2: -1.6142
With a=1 b=4 c=4
Single root: -2
```

In this exercise you can return a boolean to indicate ‘no roots’, but a boolean can have two values, and only one has meaning. For such cases there is the `monostate` class, also from the `variant` header. You would then return an object of that class as:

```
return std::monostate{};
```

#### 24.6.4.1 Calling the same function on all variants

Suppose you have a variant of some classes, which all support a method with identically signature:

```
class x_type {
public: r_type method() { ... };
};

class y_type {
public: r_type method() { ... };
};
```

It is not directly possible to call this method on a variant:

```
variant< x_type, y_type> xy;
// WRONG xy.method();
```

For a specific example we def classes `double_object`, `string_object` containing a string and a double respectively, and both classes have methods `stringer` that gives a string representation, and `sizer` that gives the ‘size’ of the object. Next we define two variant objects, one of each type. We also define *functors*

String object class:

```
// union/visit.cpp
class string_object {
private:
  string s;
public:
  string_object( string s );
  string value();
};
```

Variant objects:

```
// union/visit.cpp
variant<double_object, string_object>
union_is_double{ double_object(2.5) },
union_is_string{ string{"two-point-five"} };
```

In order to apply the method `string` to a variant of the two types we need `visit` from the `variant` header. We make a functor: a class with an overloaded `operator()`. This is used to apply an object (defined below) to the variant object:

**Code:**

```
// union/visit.cpp
cout << "Size of <<" 
    << visit( stringer{},union_is_double )
    << ">> is "
    << visit( sizer{},union_is_double )
    << '\n';
cout << "Size of <<" 
    << visit( stringer{},union_is_string )
    << ">> is "
    << visit( sizer{},union_is_string )
    << '\n';
```

**Output:**

```
[union] visit:
Size of <<2.5>> is 2
Size of <<two-point-five>>
→is 14
```

The implementations of the functors are as follows:

Stringer class:

```
// union/visit.cpp
class stringer {
public:
    string operator()
        ( double_object d ) {
        stringstream ss;
        ss << d.value();
        return ss.str(); }
    string operator()
        ( string_object s ) {
        return s.value(); }
};
```

Sizer class:

```
// union/visit.cpp
class sizer {
public:
    int operator()
        ( double_object d ) {
        return static_cast<int>( d.value() );
    }
    int operator()
        ( string_object s ) {
        return s.value().size(); }
};
```

**Remark** There is a further visit idiom:

```
template <typename... Ts>
struct overload : Ts... { using Ts::operator... ; };
f = overload {
    [] ( int i ) { /* int stuff */ };
    [] ( float f ) { /* float stuff */ };
};
```

This uses variadic templates, a topic not covered here.

### 24.6.5 Any

While `variant` can be any of a number of pre specified types, `std::any` can contain really any type. Thus it is the equivalent of `void*` in C.

An `any` object can be cast with `any_cast`:

```
std::any a{12};
std::any_cast<int>(a); // succeeds
std::any_cast<string>(a); // fails
```

## 24.7 Random numbers

The standard library has a *random number generator* that is based on a two (or three) step process, that decouples randomness from the distribution of the random numbers.

Use header:

```
#include <random>
```

Steps:

1. First there is the random engine which contains the mathematical random number generator.
2. The random numbers used in your code then come from applying a distribution to this engine.
3. Optionally, you can use a random seed, so that each program run generates a different sequence.

There several choices of engines and distributions, of which we will discuss some.

### 24.7.1 Generator

There is an implementation-defined generator `default_random_engine`:

```
std::default_random_engine generator;
```

Used this way, the generator will start at the same value every time. To seed it, use `random_device`:

```
std::random_device r;
std::default_random_engine generator{ r() };
```

This typically relies on an OS function to find a random seed.

To get reproducible random numbers, you can seed (certain) generators with a `seed` function:

```
std::default_random_engine generator;
generator.seed(5);
```

(This is especially useful in parallel, where you can seed with a process or thread number.)

Other generators include the *Mersenne twister* `mt19937`, `knuth_b`, and several others. They differ in how much space they need to maintain their internal state, and the length of their period.

### 24.7.2 Distributions

The most common mode of generating random number is to pick them from a distribution. For instance, a uniform distribution between given bounds:

```
std::uniform_real_distribution<float> distribution(0.,1.);
```

```
Code:
// rand/xrand.cpp
// seed the generator
std::random_device r;
// set the default random number generator
std::default_random_engine generator{r()};
// distribution: real between 0 and 1
std::uniform_real_distribution<float>
    distribution(0.,1.);

for ( int i=0; i<5; i++)
    cout << "random: "
        << distribution(generator)
        << '\n';
```

```
Output:
[rand] xrand:
random: 0.38229
random: 0.140263
random: 0.400579
random: 0.535661
random: 0.847095
```

A roll of the dice would result from:

```
std::uniform_int_distribution<int> distribution(1,6);
```

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
// generates number in the range 1..6
```

Other distributions are Poisson, Bernoulli, Normal, Chi-squared, and several more.

Poisson distributed integers:  
chance of  $k$  occurrences, if  $m$  is the average number  
(or  $1/m$  the probability)

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

### 24.7.3 Usage scenarios

In some scenarios you want, as it were, a bunch of random number generators. For example, in a simulation you could have multiple objects that each take random actions.

```
class Thing {
public:
    void do_something_random() {
        rnd = some_distribution( some_generator );
        f(rnd);
    };
};
```

```
};

int main() {
    for (many iterations) {
        Thing t;
        t.something_random();
    }
}
```

You might be tempted not include not only the invocation of a Random Number Generator (RNG) in a routine that needs it, but also the definition.

Wrong approach: random generator local in the function.

**Code:**

```
// rand/static.cpp
int nonrandom_int(int max) {
    std::default_random_engine engine;
    std::uniform_int_distribution<>
        ints(1,max);
    return ints(engine);
}
/* ... */
// call 'nonrandom_int' three times
```

**Output:**

[rand] nonrandom:

Three ints: 15, 15, 15.

Generator gets recreated in every function call.

This is not going to work, because the generator will be initialized every time you call the function, leading to a sequence of identical numbers.

**Exercise 24.10.** What is wrong with the following code:

```
int somewhat_random_int(int max) {
    random_device r;
    default_random_engine generator{ r() };
    std::uniform_int_distribution<> ints(1,max);
    return ints(generator);
};
```

You can fix this by making the generator variable `static`, so that there is only one generator, shared by all function invocations.

Good approach: random generator static in the function.

**Code:**

```
// rand/static.cpp
int realrandom_int(int max) {
    static std::default_random_engine
        static_engine;
    std::uniform_int_distribution<>
        ints(1,max);
    return ints(static_engine);
}
```

**Output:**

[rand] truerandom:  
Three ints: 15, 98, 70.

A single instance is ever created.

**Remark** A similar diagnosis and solution hold for objects:

```
class Thing {
private:
    random_device r;
    default_random_engine generator{ r() };
public:
    void do_something_random() {
        rnd = some_distribution( generator );
        f(rnd);
    };
};
```

The solution is that the device and generator should be static.

What if you have multiple routines that use random numbers? To keep it all statistically justified you could move the RNG into a separate routine:

```
// rand/static.cpp
int realrandom_int(int max) {
    static std::default_random_engine static_engine;
    std::uniform_int_distribution<>
        ints(1,max);
    return ints(static_engine);
}
```

To clean up this design, you could then even put this in an object with inline static class methods:

**Note the use of `static`:**

```
// rand/randname.cpp
class generate {
private:
    static inline std::default_random_engine engine;
public:
    static int random_int(int max) {
        std::uniform_int_distribution<> ints(1,max);
        return ints(generate::engine);
    };
};
```

**Usage:**

```
auto nonzero_percentage = generate::random_int(100)
```

#### 24.7.4 Permutations

The function `shuffle` shuffles an array. Coupled with the `iota` function (from the `numeric` header) this easily gives a permutation:

**Code:**

```
// rand/shuffle.cpp
std::vector<int> idxs(20);
iota(idxs.begin(), idxs.end(), 0);
/* ... */
std::shuffle
    (idxs.begin(), idxs.end(), g);
```

**Output:**

**[rand] shuffle:**

0	1	2	3	4	5	6
↑	7	8	9			
10	11	12	13	14	15	16
↑	17	18	19			

**Iota:**

6	9	4	3	16	15	18
↑	5	2	11			
14	19	17	1	0	13	7
↑	10	12	8			

**Permute:**

6	9	4	3	16	15	18
↑	5	2	11			
14	19	17	1	0	13	7
↑	10	12	8			

(where `g` is a random generator.)

#### 24.7.5 C random function

There are several *random number generators* available in the standard library of the C language. These should no longer be used for C++ code.

They come with threadsafe-variants where the state is explicitly passed, and therefore it can be stored in thread-private storage.

- `rand48`, `rand48_r` a 48-bit multiplicative generator and a threadsafe variant;
- `rand`, `rand_r` by far the most common;
- `random`, `random_r`: a better generator, compliant with recent *POSIX* standards.

The function `rand` yields an `int` – a different one every time you call it – in the range from zero to `RAND_MAX`. Using scaling and casting you can then produce a fraction between zero and one with the above code.

```
#include <random>
using std::rand;
float random_fraction =
    (float)rand() / (float)RAND_MAX;
```

This generator has some problems.

- The C random number generator has a period that can be as small as  $2^{15}$ .
- There is only one generator algorithm, which is implementation-dependent, and has no guarantees on its quality.
- There are no mechanisms for transforming the sequence to a range. The common idiom

```
int under100 = rand() % 100
```

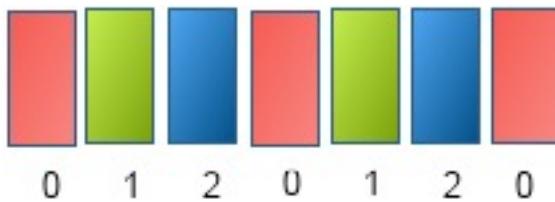


Figure 24.1: Low number bias of a random number generator taken module

is biased to small numbers. Figure 24.1 shows this for a generator with period 7 taken modulo 3.

If you run your program twice, you will twice get the same sequence of random numbers. That is great for debugging your program but not if you were hoping to do some statistical analysis. Therefore you can set the *random number seed* from which the random sequence starts by the `srand` function. Example:

```
srand(time(NULL));
```

seeds the random number generator from the current time. This call should happen only once, typically somewhere high up in your main.

## 24.8 Time

We will mostly devote attention to measurement of time intervals; since this is most relevant to scientific computing. Section 24.8.4 will briefly go into functions for expression absolute time.

Header

```
#include <chrono>
```

As a simple example how you would time a function `f`:

```
// chrono/simple.cpp
using clock = std::chrono::steady_clock;
clock::time_point before = clock::now();
f();
auto after = clock::now();
cout << "Duration: "
<< std::chrono::duration_cast<std::chrono::milliseconds>(after-before).count()
<< "ms\n";
```

### 24.8.1 Time durations

You can define durations with `seconds`:

```
// chrono/second.cpp
seconds s{3};
auto t = 4s;
```

You can do arithmetic and comparisons on this type:

**Code:**

```
// chrono/second.cpp
cout << "This lasts "
    << s.count() << "s" << '\n';
cout << "This lasts ";
print_seconds( s+5s );
auto nine = 3.14*3s;
cout << nine.count()
    << "s is under 10 sec: "
    << boolalpha << (nine<10s)
    << '\n';
```

**Output:**

```
[chrono] basicsecond:
This lasts 3s
This lasts 8s
9.42s is under 10 sec: true
```

While `seconds` takes an integer argument, you can then multiply or divide it to get fractional values.

The `millisecond` function gives a duration, and you can convert seconds implicitly to milli, but the other way around you need `duration_count`:

```
// chrono/second.cpp
print_milliseconds( 5s );
// DOES NOT COMPILE print_seconds( 6ms );
print_seconds( duration_cast<seconds>(6ms) );
```

The full list of durations (with suffixes) is: `hours` (`1h`), `minutes` (`1min`), `seconds` (`1s`), `milliseconds` (`1ms`), `microseconds` (`1us`), `nanoseconds` (`1ns`).

### 24.8.2 Time points

A *time point* can be considered as a duration from some starting point, such as the start of the Unix epoch: the start of the year 1970.

```
time_point<system_clock, seconds> tp{10'000s};
```

is  $2h + 46\text{min} + 40s$  into 1970.

You make this explicit by calling the `time_since_epoch` method on a time point, giving a duration.

### 24.8.3 Clocks

There are several clocks. The common supplied clocks are

- `system_clock` for time points that have a relation to the calendar; and
- `steady_clock` for precise measurements.
- C++20 added some more clocks like `utc_clock`, that are like `system_clock` but with a different offset.

Usually, `high_resolution_clock` is a synonym for either of these, but because of this lack of precise definition you should probably not use it.

A clock has methods:

- `duration` indicates a time duration; it consists of `rep`, the repetition count, and a `period` which is usually 1.

- `time_point` representing a point in time. This is usually obtained from the `now` method of a clock:

```
const std::chrono::time_point<std::chrono::system_clock> now =
    std::chrono::system_clock::now();
```

- Some clocks obey leap years and leap seconds; the property `is_steady` indicates whether a clock makes the time progress at one second per second, that is, without leaps.

As you saw above, a `time_point` is associated with a clock, and time points of different clocks can not be compared or converted to each other.

#### 24.8.3.1 Duration measurement

To time a segment of execution, use the `now` method of the clock, before and after the segment. Subtracting the time points gives a duration in nanoseconds, which you can cast to anything else by `duration_cast`:

<b>Code:</b>	<b>Output:</b>
<pre>// chrono/clock.cpp using myclock = system_clock; myclock::time_point before =     myclock::now(); // do something that takes time std::this_thread::sleep_for( 1.5s );// snippetexlude auto after = myclock::now(); cout &lt;&lt; "Action took: "     &lt;&lt; duration_cast&lt;milliseconds&gt;         (after-before).count()     &lt;&lt; "ms\n";</pre>	<b>[chrono] clock:</b> <i>Slept for 1503ms</i>

#### 24.8.3.2 Clock resolution

The `clock resolution` can be found from the `period` property:

```
auto
num = myclock::period::num,
den = myclock::period::den;
auto tick = static_cast<double>(num) / static_cast<double>(den);
```

Timing:

```
auto start_time = myclock::now();
auto duration = myclock::now() - start_time;
auto microsec_duration =
    std::chrono::duration_cast<std::chrono::microseconds>(duration);
cout << "This took " << microsec_duration.count() << "usec" << endl;
```

Computing new time points:

```
auto deadline = myclock.now() + std::chrono::seconds(10);
```

#### 24.8.4 Dates

Pre-C++20, `chrono` was only concerned with time durations; in C++20 utilities for calendar and time-of-day were added.

TODO some examples

#### 24.8.5 C mechanisms not to use anymore

Letting your process sleep: `sleep`

Time measurement: `getrusage`

### 24.9 File system

As of the C++17 standard, there is a file system library `filesystem`, which includes things like a directory walker.

```
#include <filesystem>
```

### 24.10 Enum classes

The C-style `enum` keyword defines global names, so it's easy to have name collisions. For example:

```
// enum/enumnot.cpp
enum trafficlight { red, yellow, green };
cout << red << "," << yellow << "," << green << '\n';
enum flag { red, white, blue }; // Collision!
```

In C++ the `enum class` (or `enum struct`) is introduced, which makes the names into class members. For instance, you can make a vector of such enum class objects and fill it:

```
// enum/enumclass.cpp
enum class flowers { grass, poppy, bluebonnet };
vector<flowers> field(10, flowers::grass);
field[1] = flowers::poppy;
field[5] = flowers::bluebonnet;
```

With the enum class mechanism you can now have multiple identical names for an enum, since they are scoped to a class (C++ Core Guidelines [8], Enum.3).

By default, a variable of an enum type is allocated as an `int`. You can change this by letting the type inherit from another type:

```
// enum/enumclass.cpp
enum class flag : unsigned short { red, white, blue };
```

If such a class inherits from an integral type, this does not mean it behaves like an integer; for instance, you can not immediately ask if one is less than another. Instead, this form of inheritance only determines the amount of space taken for an enum item.

To let enum objects behave like the underlying integral type, you need to cast it:

```
// enum/enumclass.cpp
cout << static_cast<unsigned short>( flag::red ) << ","
    << static_cast<unsigned short>( flag::white ) << ","
    << static_cast<unsigned short>( flag::blue ) << '\n';
```

In C++23 this can be done more concisely with `to_underlying`:

```
// enum/enumclass.cpp
enum class flag : unsigned int { red, white, blue };
// but we still need to cast them
cout << std::to_underlying( flag::red ) << ","
    << std::to_underlying( flag::white ) << ","
    << std::to_underlying( flag::blue ) << '\n';
```

Instead of an `enum class` you can often also enclose your `enum` in the namespace of a class:

```
// enum/spaced.cpp
class Field {
public:
    enum color { invalid=-1, white=0, red=1, blue=2, green=3 };
private:
    color mycolor;
public:
    void set_color( color c ) { mycolor = c; }
    /* ... */
Field onefield;
onefield.set_color( Field::color::blue );
```

## 24.11 Orderings and the ‘spaceship’ operator

It is possible to overload comparison operators `<`, `<=`, `==`, `>=`, `>`, `!=` for classes; section 9.5.7. However, there is usually a lot of redundancy in doing so, even if you express one operator in terms of another:

```
bool MyClass::operator>=( const MyClass& other ) {
    return not (other<*this);
};
```

The C++20 `spaceship operator` `<=>` can make life a lot easier. This binary operator returns an ordering object, which can be one of the six relations.

Let’s illustrate with a simple example. Each object of the class `Record` has a string and a unique number:

```
// stl/spaceship.cpp
class Record {
private:
    static inline int nrecords{0};
    int myrecord{-1};
    string name;
public:
    Record( string name )
        : name(name) { myrecord = nrecords++; }
```

Now you want an ordering relation purely based on the record number.

**Code:**

```
// stl/spaceship.cpp
Record alice("alice"), bob("bob");
cout.expect(t f t t f f);
cout.println("{}", (alice==alice));
cout.println("{}", (alice==bob));
cout.println("{}", (alice<=bob));
cout.println("{}", (alice<bob));
cout.println("{}", (alice>=bob));
cout.println("{}", (alice>bob));
```

**Output:**

```
[stl] spacerecord:
expect t f t t f f
true
false
true
true
false
false
```

We implement this relation using the spaceship operator, using the fact that an ordering on integers exists:

```
// stl/spaceship.cpp
std::strong_ordering operator<=> ( const Record& other ) const {
    return myrecord<=>other.myrecord;
}
bool operator==( const Record& other) const {
    return myrecord==other.myrecord;
};
```

The return type of this comparison is a `std::strong_ordering`, because in this case any two objects  $x, y$  always satisfy exactly one of

$x < y$     $x == y$     $x > y$

For a more complicated example, let's make a `Coordinate` class, where two coordinates compare as  $<, =, >$  if all components satisfy that relation. This is a partial ordering, meaning that some coordinates  $x, y$  do not satisfy any of

$x < y$ ,  $x == y$ ,  $x > y$

**Code:**

```
// stl/spaceship.cpp
Coordinate<int> p12(1,2), p24(2,4), p31(3,1);
cout.expect(t t f);
cout.println("{}",(p12==p12));
cout.println("{}",(p12<p24));
cout.println("{}",
(p12<p31 or p12>p31 or p12==p31));
```

**Output:**

```
[stl] spacepartial:
expect t t f
true
true
false
```

In this case the spaceship operator returns a `partial_ordering` result. Equality needs to be defined separately again:

```
// stl/spaceship.cpp
std::partial_ordering operator<=> ( const Coordinate& other ) const {
    std::strong_ordering c = components[0] <=> other.components[0];
    for (int i = 1; i < components.size(); ++i) {
        if ((components[i] <=> other.components[i]) != c)
            return std::partial_ordering::unordered;
```

```
    }
    return c;
};

bool operator==( const Coordinate& other ) const {
    for ( int ic=0; ic<components.size(); ++ic )
        if (components[ic] !=other.components[ic])
            return false;
    return true;
};
```



# Chapter 25

## Concurrency

Concurrency is a difficult subject. It is both like and unlike parallelism, which is a prime concern on modern scientific computing, so it is both relevant, and not-so relevant, to the target audience of this book.

In a way, parallelism is clear: it's about things happening at the same time, for instance on the multiple cores of your modern processor. The benefit of parallelism is that your code runs faster, and the challenge is how to write your program to express that there are indeed things that can be done simultaneously.

Etymologically, ‘concurrency’ also seems to be about things happening simultaneously. However, it predates actual parallel processors: concurrency is important in OSs where even on a single processor you have a program running, simultaneously with processes that are printing, checking mail, et cetera.

Thus, concurrency can be defined as the study of activities that have no clear temporal relation. The main issue of study is then how to deal with ‘shared state’: objects that are accessed in no clear order by more than one process.

Concurrency is important in such areas as OSs, *databases*, and *web servers*. The main mechanism for concurrency is threading, which we cover in this chapter. The C++20 standard also added *co-routines*, which we do not cover here.

For scientific computing, parallelism is more important than concurrency; for this we refer to Volume 2 of this series, which covers the *MPI* and *OpenMP* systems, both of which can easily be used from C++.

### 25.1 Thread creation

This uses the *thread* header:

```
#include <thread>
```

A thread is an object of class `std::thread`, and creating it you immediately begin its execution. The thread constructor takes at least one argument, a *callable* (a function pointer or a lambda), and optionally the arguments of that function-like object.

The environment that calls the thread needs to call *join* to ensure the completion of the thread. Until you do so, there is no guarantee that the thread’s function has run.

As a very simple example, here we have a thread that sleeps for a second. Note: this uses the OS *sleep* function; there are better mechanisms; see section 25.1.4.

**Code:**

```
// thread/block.cpp
auto start_time = Clock::now();
auto waiting_thread =
    std::thread( []() {
        sleep(1);
    });
waiting_thread.join();
auto duration = Clock::now()-start_time;
```

**Output:**

```
[thread] block:
This took 1.00136 sec
```

An example with a function that takes arguments:

```
#include <thread>

auto somefunc = [] (int i,int j) { /* stuff */ };
std::thread mythread( somefunc, arg1,arg2 );
mythread.join()
```

### 25.1.1 Multiple threads

Creating a single thread is not very useful. Often you will create multiple threads that will subdivide work, and then wait until they all finish.

```
vector<thread> mythreads;
for ( i=/* stuff */ )
    mythreads.push_back( thread( somefunc, someargs(i) ) );
for ( i=/* stuff */ )
    mythreads[i].join();
```

Note that we create the vector of threads with size zero, and use `push_back` to fill it, since a thread starts executing. Creating the vector immediately with the right size would have created a bunch of no-op threads.

**Exercise 25.1.** If you are very concerned about the cost of allocation, how can you create the space for the threads, without creating the threads.

Here is a simple hello world example. Because no guarantee exist on the ordering of when threads start or end, the output can look messy:

**Code:**

```
// thread/hello.cpp
vector< std::thread > threads;
for ( int i=0; i<NTHREADS-1; ++i ) {
    threads.push_back
        ( std::thread(hello_n, i) );
}
threads.emplace_back
    ( hello_n, NTHREADS-1 );
for ( auto& t : threads )
    t.join();
```

**Output:**

```
[thread] hellomess:
Hello Hello 01
Hello 2
Hello 3
Hello 4
```

(Note the call to `emplace_back`: because of *perfect forwarding* it can invoke the constructor on the `thread` arguments.)

We bring order in this message by including a wait. As a separate issue, the thread now executes a function given by a lambda expression:

```
Code:
// thread/hello.cpp
threads.push_back
( std::thread
( /* function: */
[] (int i) {
    std::chrono::seconds wait(i);
    std::this_thread::sleep_for(wait);
    hello_n(i);
    /* argument: */
    i
}
);
```

```
Output:
[thread] hellonice:
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
```

Of course, in practice you don't synchronize threads through waits. Read on.

### 25.1.2 Asynchronous tasks

One problem with threads is how to return data. You could solve that with capturing a variable by reference, but that is not very elegant. A better solution would be if you could ask a thread ‘what is the thing you just calculated’.

For this we have the `std::future`, from the header `future`, templated with the return type. Thus, a `std::future<int>`

will be an `int`, somewhere in the future. You retrieve the value with `get`:

```
// thread/async.cpp
std::future<string> fut_str = std::async
( [] () -> string { return "Hello world"; } );
auto result_str = fut_str.get();
cout << result_str << '\n';
```

An example with multiple futures:

```
// thread/async.cpp
vector<std::future<string>> futures;
for ( int ithread=0; ithread<NTHREADS; ++ithread ) {
    futures.push_back
    ( std::async
        ( [ithread] () ->string {
            stringstream ss;
            ss << "Hello world " << ithread;
            return ss.str();
        } ) );
}
```

```
for ( int ithread=0; ithread<NTHREADS; ++ithread ) {
    cout << futures.at(ithread).get() << '\n';
}
```

One problem with `async` is that the task need not execute on a new thread: the runtime can decide to execute it on the calling thread, only when the `get` call is made. To force a new thread to be spawned immediately, use

```
auto fut = std::async
    ( std::launch::async, fn, arg1, arg2 );
```

### 25.1.3 Return results: futures and promises

Explicit use of promises and futures is on a lower level than `async`.

```
// thread/promise.cpp
auto promise = std::promise<std::string>();
auto producer = std::thread
    ( [&promise] { promise.set_value("Hello World"); } );

auto future = promise.get_future();

auto consumer = std::thread
    ( [&future] { std::cout << future.get() << '\n'; } );

producer.join(); consumer.join();

// thread/promise.cpp
vector< std::thread > producers, consumers
;
vector< std::promise<string> > promises;

for ( int i=0; i<4; ++i ) {

    promises.push_back( std::promise<
        string>() );
    producers.push_back
        ( std::thread
            ( [ i,&promises ] {
                stringstream ss;
                ss << "Hello world " << i << ".";
                promises.at(i).set_value(ss.str())
            } ) );
}

consumers.push_back
    ( std::thread
        ( [ i,&promises ] {
            std::cout << promises.at(i).
            get_future().get() << '\n';
        } ) );
for ( auto& p : producers ) p.join();
for ( auto& c : consumers ) c.join();
```

### 25.1.4 The current thread

```
std::this_thread::get_id();
```

This is a unique ID, but you can not derive any further information from it. For instance, it is not related to OS process IDs, or the number of cores of your processor.

To put a thread to sleep, use `sleep_for`:

```
// thread/hello.cpp
threads.push_back
( std::thread
( /* function: */
[] (int i) {
    std::chrono::seconds wait(i);
    std::this_thread::sleep_for(wait);
    hello_n(i); },
/* argument: */
i
)
);
```

### 25.1.5 More thread stuff

The C++20 `jthread` launches a thread which will join when its destructor is called. With the creation loop:

```
{ // start a scope
vector<thread> mythreads;
for ( i=/* stuff */ )
    mythreads.push_back( thread( somefunc, someargs ) );
} // end of scope
```

The joins now happen without an explicit call when the vector goes out of scope. This also solves the problem that explicit `join` calls are ordered.

## 25.2 Data races

An important topic in concurrency is that of *data races*: the phenomenon that multiple accesses to a single data item are not temporally or causally ordered, for instance because the accesses are from threads that are simultaneously active.

```
std::mutex alock;
alock.lock();
/* critical section */
alock.unlock();
```

This has a bunch of problems, for instance if the critical section can throw an exception.

One solution is `std::lock_guard`:

```
std::mutex alock;
thread( [] () {
    std::lock_guard<std::mutex> myguard(alock);
    /* critical stuff */
} );
```

The lock guard locks the mutex when it is created, and unlocks it when it goes out of scope.

For C++17, `std::scoped_lock` can do this with multiple mutexes.

Atomic variables exist in the `atomic` header:

```
#include <atomic>
std::atomic<int> shared_int;
shared_int++;
```

Communication between threads:

```
std::condition_variable somecondition;
// thread 1:
std::mutex alock;
std::unique_lock<std::mutex> ulock(alock);
somecondition.wait(ulock)
// thread 2:
somecondition.notify_one();
```

Similar but different:

```
#include <future>
std::future<int> future_computation =
    std::async( [] (int x) { return f(x); },
               100 );
future_computation.get();

std::future_status comp_status;
comp_status = future_computation.wait_for( /* chrono duration */ );
if (comp_status==std::future_status::ready)
    /* computation has finished */
```

### 25.3 Synchronization

Threads run concurrent with the environment (or thread) that created them. That means that actions have no temporal ordering prior to synchronization with `std::thread::join`.

In this example, the concurrent updates of `counter` form a *data race*:

<b>Code:</b>	<b>Output:</b>
<pre>// thread/race.cpp auto start_time = myclock::now(); auto deadline = myclock::now() + std::     chrono::seconds(1); int counter{0}; auto add_thread =     std::thread( [&amp;counter, deadline] () {         while (myclock::now()&lt;deadline)             printf("Thread: %d\n", ++counter);     }); while (myclock::now()&lt;deadline)     printf("Main: %d\n", ++counter); add_thread.join(); cout &lt;&lt; "Final value: " &lt;&lt; counter &lt;&lt; '\n' ';</pre>	<b>[thread] race:</b> <p>Three runs of &lt;&lt;race&gt;&gt;; printing first lines only: ----</p> <p>Main: 1 Thread: 51 Final value: 526851 Runtime: 1.00048 sec ----</p> <p>Main: 1 Thread: 47 Final value: 617669 Runtime: 1.00243 sec ----</p> <p>Main: 1 Thread: 47 Final value: 509073 Runtime: 1.00215 sec</p>

Formally, this program has Undefined Behavior (UB), which you see reflected in the different final values.

The final value can be fixed by declaring the counter as `std::atomic`:

**Code:**

```
// thread/atomic.cpp
auto start_time = myclock::now();
auto deadline = myclock::now() + std::
    chrono::seconds(1);
std::atomic<int> counter{0};
auto add_thread =
    std::thread( [&counter, deadline] () {
        while (myclock::now() < deadline)
            printf("Thread: %d\n", ++counter);
    });
while (myclock::now() < deadline)
    printf("Main: %d\n", ++counter);
add_thread.join();
cout << "Final value: " << counter << '\n'
';
```

**Output:**

[`thread`] `atomic`:

```
Three runs of <<atomic>>;
printing first lines only:
-----
Main: 1
Thread: 54
Final value: 495120
Runtime: 1.00282 sec
-----
Thread: 1
Main: 353
Final value: 474618
Runtime: 1.00312 sec
-----
Main: 1
Thread: 59
Final value: 339453
Runtime: 1.00212 sec
```

Note that the accesses by the main and the thread are still not predictable, but that is a feature, not a bug, and definitely not UB.



# Chapter 26

## Obscure stuff

### 26.1 Auto

The `auto` keyword can be used in circumstances where the compiler can deduce the type of a variable, saving you from having to spell it out.

#### 26.1.1 Declarations

A first use of `auto` is in declarations. Sometimes the type of a variable being declared is obvious from the initial value it is assigned:

```
std::vector< std::shared_ptr< myclass >>*
myvar = new std::vector< std::shared_ptr< myclass >>
( 20, new myclass(1.3) );
```

(Pointer to vector of 20 shared pointers to `myclass`, initialized with unique instances.) You can write this as:

```
auto myvar =
new std::vector< std::shared_ptr< myclass >>
( 20, new myclass(1.3) );
```

Function return type deduction:

```
// auto/autofun.cpp
auto equal(int i,int j) {
    return i==j;
};
```

Similar to functions, the return type of class methods can also be declared `auto`.

Return type of methods can be deduced:

```
// auto/plainget.cpp
class A {
private: float data;
public:
A(float i) : data(i) {};
auto &access() {
    return data; };
void print() {
```

## 26. Obscure stuff

---

```
    cout << "data: " << data << '\n'; };
```

If you declare a variable with `auto`, and you assign a reference to it, the variable does not automatically become a reference:

With this class definition:

```
class X {  
private:  
    int _x;  
public:  
    int &x () { return _x};  
}
```

this is not a reference:

```
X x;  
auto ix = x.x();  
  
but this is:  
auto& ix = x.x();
```

Demonstrating that `auto` discards references from the rhs:

**Code:**

```
// auto/plainget.cpp  
A my_a(5.7);  
// reminder: float& A::access()  
auto get_data = my_a.access();  
get_data += 1;  
my_a.print();
```

**Output:**

```
[auto] plainget:  
data: 5.7
```

Combine `auto` and references:

**Code:**

```
// auto/refget.cpp  
A my_a(5.7);  
auto &get_data = my_a.access();  
get_data += 1;  
my_a.print();
```

**Output:**

```
[auto] refget:  
data: 6.7
```

For good measure:

```
// auto/constrefget.cpp  
A my_a(5.7);  
const auto &get_data = my_a.access();  
get_data += 1; // WRONG does not compile  
my_a.print();
```

### 26.1.2 `decltype`: declared type

There are places where you want the compiler to deduce the type of a variable, but where this is not immediately possible. Suppose that in

```
auto v = some_object.get_stuff();  
f(v);
```

you want to put a `try ... catch` block around just the creation of `v`. This does not work:

```
try { auto v = some_object.get_stuff();  
} catch (...) {}  
f(v);
```

because the `try` block is a scope. It also doesn't work to write

```
auto v;  
try { v = some_object.get_stuff();  
} catch (...) {}  
f(v);
```

because there is no indication what type `v` is created with. Also, this presumes the existence of a default constructor.

Instead, it is possible to query the type of the expression that creates `v` with `decltype`:

```
decltype(some_object.get_stuff()) v;  
try { auto v = some_object.get_stuff();  
} catch (...) {}  
f(v);
```

## 26.2 Casts

In C++, constants and variables have clear types. For cases where you want to force the type to be something else, there is the *cast* mechanism. With a cast you tell the compiler: treat this thing as such-and-such a type, no matter how it was defined.

**Remark** The C cast has a syntax of:

```
sometype x;  
othertype y = (othertype)x;
```

*One practical problem with this syntax is that it is hard to spot, for instance by searching in an editor. Because C++ casts have a unique keyword, they are easier to recognize in a text editor.*

We will now discuss some casting mechanisms. See section 24.6.5 for `any_cast`.

### 26.2.1 Static cast

One use of casting is to convert constants to a ‘larger’ type. For instance, allocation does not use integers but `size_t`.

## 26. Obscure stuff

---

**Code:**

```
// cast/longint.cpp
int hundredk = 100000;
int overflow =
    hundredk*hundredk;
cout << "overflow: "
    << overflow << '\n';
size_t bignum =
    static_cast<size_t>(hundredk) * hundredk;
cout << "bignum: "
    << bignum << '\n';
```

**Output:**

```
[cast] longint:
```

However, if the conversion is possible, the result may still not be ‘correct’.

**Code:**

```
// cast/intlong.cpp
long int hundredg = 1000000000000;
cout << "long number: "
    << hundredg << '\n';
int overflow;
overflow = static_cast<int>(hundredg);
cout << "assigned to int: "
    << overflow << '\n';
```

**Output:**

```
[cast] intlong:
```

There are no runtime tests on static casting.

In templated code, a static cast can be used to ensure that constants are of the right type. Suppose we have

```
g(int,int);
g(long,long);
```

then

```
long x;
g(x,0);
```

is ambiguous. Instead, use a cast to ensure a zero of the right type:

```
template< typename scalar >
void f( scalar x) {
    g( x, static_cast<scalar>(0) );
}
```

Static casts are a good way of casting back void pointers to what they were originally. This mechanism is used extensively in C codes, where void pointers are also called *opaque handles*.

Main:

```
f( (void*)(&f), 2 );
```

```
// cast/opaque.cpp
int i=5;
f( (void*)(&i), 1 );
float f=.5;
```

Function:

```
// cast/opaque.cpp
void f( void *p, int type ) {
    if (type==1) {
        int *i = static_cast<int*>(p);
        cout << i << '\n';
    } // other cases omitted
}
```

## 26.2.2 Dynamic cast

The `dynamic_cast` can be used to cast up and down a class hierarchy. In particular, it can cast pointers and references from a base to a derived class. Note that this involves a runtime check, so this type of cast is more expensive than others that fulfill a more syntactic function.

Consider the case where we have a base class and derived classes.

```
// cast/toderived.cpp
class Base {
public:
    virtual void print() = 0;
};

class Derived : public Base {
public:
    virtual void print() {
        cout << "Call Derived function!"
        << '\n'; }
};

class Eriived : public Base {
public:
    virtual void print() {
        cout << "Call Eriived function!"
        << '\n'; }
};
```

Also suppose that we have a function that takes a pointer or reference to the base class:

```
void f( Base& obj );
```

The function can discover what derived class the base pointer refers to:

```
// cast/toderived.cpp
void f( Base& obj ) {
    try {
        Derived &der =
            dynamic_cast<Derived&>(obj);
        der.print();
    } catch (...) {
        cout << "Could not be cast to Derived"
        << '\n';
    }
}
```

If the dynamic cast is unsuccessful:

- on pointer, it gives a *null pointer*;
- on a reference, throws a `bad_cast` exception.

If we have a pointer to a derived object, stored in a pointer to a base class object, it's possible to turn it safely into a derived pointer again:

```
Code:
// cast/toderived.cpp
cout << "Dynamic cast to Derived\n";
Derived object1;
f(object1);
Eriived object2;
f(object2);
```

```
Output:
[cast] deriveright:
Construct derived!
Could not be cast to Derived
```

On the other hand, a `static_cast` would not do the job:

```
Code:
// cast/toderiveddp.cpp
Base *object = new Derived();
g(object);
Base *nobject = new Eriived();
g(nobject);
```

```
Output:
[cast] derivewrong:
Construct derived!
Construct erived!
```

Note: the base class needs to be polymorphic, meaning that a pure virtual method is needed. This is not the case with a static cast, but, as said, this does not work correctly in this case.

### 26.2.3 Const cast

With `const_cast` you can add or remove const from a variable. This is the only cast that can do this.

This cast can go two ways, and one is considered a bad idea: if something is marked const and you want to alter it, there is something wrong with your design (C++ Core Guidelines [8], ES.50). You probably need to use `mutable`; section 18.6.

However, there is a use case for the other direction. Suppose you are interfacing to a library that is not strict about its use of `const`, as if for instance the case with many old C libraries. Then you may need to cast some arguments to const when you call this function from a const method of your own.

In this example, both the input and output argument of a C routine are not const. To pass a `const` array to this routine, you need to `const_cast` the `const` away.

```
// const/constcast.cpp
void old_c_routine( float*in, float*out );
void new_cpp_routine( const vector<float>&in, vector<float>&out ) {
    old_c_routine
        ( const_cast<float*>( in.data() ), out.data() );
}
```

As a less verbose alternative to `const_cast`, you can use the (C++17) templated function `as_const`.

### 26.2.4 Reinterpret cast

The `reinterpret_cast` and `std::bit_cast` (in the `bit` header; C++20) perform the functions corresponds to the C mechanism of ‘take this byte and pretend that it is of type whatever’. The former is mostly intended for conversions to and from pointers:

---

```
int i = 7;
char* p2 = reinterpret_cast<char*>(&i);
std::cout << (p2[0] == '\x7' ? "This system is little-endian\n"
               : "This system is big-endian\n");
```

### 26.2.5 A word about void pointers

A traditional use for casts in C was the treatment of *void pointers*. The need for this is not as severe in C++ as it was before.

A typical use of void pointers appears in the PETSc [3, 4] library. Normally when you call a library routine, you have no further access to what happens inside that routine. However, PETSc has the functionality for you to specify a monitor so that you can print out internal quantities.

```
int KSPSetMonitor(KSP ksp,
    int (*monitor)(KSP, int, PetscReal, void*),
    void *context,
    // one parameter omitted
);
```

Here you can declare your own monitor routine that will be called internally: the library makes a *callback* to your code. Since the library can not predict whether your monitor routine may need further information in order to function, there is the *context* argument, where you can pass a structure as void pointer.

This mechanism is no longer needed in C++ where you would use a *lambda*:

```
KSPSetMonitor( ksp,
    [mycontext] (KSP k, int , PetscReal r) -> int {
        my_monitor_function(k, r, mycontext); } );
```

## 26.3 Fine points of scalar types

### 26.3.1 Booleans

#### 26.3.1.1 Vector of *bool*

Watch out for `vector<bool>`: this uses a compact scheme storing multiple values per byte. This has the unfortunate side-effect that you can not get a reference to a vector element. Thus, while you can write

```
for ( auto b : my_vector_of_bool ) /* something */
```

writing the following is not grammatically correct:

```
// DOES NOT COMPILE
for ( auto& b : my_vector_of_bool ) /* something */
```

Also, multiple threads may not be able to set elements independently because of *false sharing*.

### 26.3.2 Integers

There are several integer types, differing by the number of bytes they are stored in, and by whether they are signed or not. Here is a systematic discussion.

### 26.3.2.1 The long and short of it

In addition to `int`, there are also `short`, `long` and `long long` integers. These names give some imperfect indication as to their size, and therefore range. The actual sizes and ranges of these types are implementation defined. The next section will discuss types with a more precise definition.

- A `short int` is at least 16 bits (if you need an 8-bit integral datatype, use `char`);
- An `int` is at least 16 bits, which was the case in the old days of the *DEC PDP-11*, but nowadays they are commonly 32 bits;
- A `long int` is at least 32 bits, but often 64; the big exception being the *Windows OS*, where a `long` is 32 bits.
- A `long long int` is at least 64 bits.

There are a number of generally accepted *data models* for the definition of these types; see HPC book [13], section 3.7.1.

All these types are signed integers: a  $k$ -bit integer accommodates a range  $-2^{k-1} \dots 0 \dots 2^{k-1} - 1$ . Prefixing these types with the keyword `unsigned` gives nonnegative types with a range  $0 \dots 2^k - 1$ .

If you want to determine precisely the range of integers (or real numbers, which we discuss later) that is stored in an integral variable you can use the `limits` header; see section 24.2.

### 26.3.2.2 Specifying the number width

If you want to specify the number bits that an integer variable should have, there is the `cstdint` header containing *fixed width integer types*. It defines such types as `int16_t` and `uint16_t`.

```
std::int8_t    // 8 bits
std::uint8_t   // 8 bits, unsigned
std::int16_t   // 16 bits
std::uint16_t  // 16 bits, unsigned
std::int32_t   // 32 bits
std::uint32_t  // 32 bits, unsigned
std::int64_t   // 64 bits
std::uint64_t  // 64 bits, unsigned
```

The following example confirms these sizes by using the `sizeof` function:

Code:	Output:
<pre>// int/sizeof.cpp int8_t i8; uint8_t u8; int16_t i16; uint16_t u16; printf("{}\n", sizeof(i8),        sizeof(u8),        sizeof(i16),        sizeof(u16)      );</pre>	<pre>[int] sizeof: 1, 1, 2, 2</pre>

Since C++20, integers are guaranteed to be stored as ‘two’s complement’; see HPC book [13], section 3.2.

The use of unsigned values is fraught with danger since they are often first converted to signed quantities. For instance, comparing them to signed integers gives counter-intuitive results:

```
Code:
// int/unsigned.cpp
unsigned int one{1};
int minusone{-1};
println("less: {}", (minusone<one));
```

```
Output:
[int] cmp:
less: false
```

For this reason, C++20 has introduced utility functions `cmp_equal`, `cmp_not_equal`, `cmp_greater` et cetera, that do these comparisons correctly.

#### 26.3.2.3 Why different integers?

Historically, short integers, defined as `short` or `short int`, were used as a way to save space when large values were not needed. These days, that argument is largely irrelevant. The argument for short integers now derives from the limited *bandwidth* of processors: ‘streaming’ type applications have a performance that is determined by the available bandwidth. Using short integers effectively doubles that bandwidth.

The need for long integers, defined as `long` or `long int`, arises from the current abundance of memory and disc space. Modern applications have large amounts of memory available to them, more than can be addressed with a 32-bit integer. This phenomenon is exacerbated by the existence of *multicore* processors, that can handle many times the memory of earlier processors in the same time. In scientific computing 64-bit integers are getting increasingly common.

#### 26.3.3 Floating point types

Truncation and precision are tricky things. As a small illustration, let’s do the same computation in single and double precision. While the results show the same with the default `cout` formatting, if we subtract them we see a non-zero difference.

```
Code:
// basic/point3.cpp
double point3d = .3/.7;
float
point3f = .3f/.7f,
point3t = point3d;
cout << "double precision: "
<< point3d << '\n'
<< "single precision: "
<< point3f << '\n'
<< "difference with truncation:"
<< point3t - point3f
<< '\n';
```

```
Output:
[basic] point3:
double precision: 0.428571
single precision: 0.428571
difference with
truncation:-2.98023e-08
```

You can actually explain the size of this difference, however, we defer discussion of the details of floating point arithmetic to HPC book [13], chapter 3.

### 26.3.4 Not-a-number

The *IEEE 754* standard for floating point numbers states that certain bit patterns correspond to the value *Nan*: ‘not a number’. This is the result of such computations as the square root of a negative number, or zero divided by zero; you can also explicitly generate it with `quiet_NaN` or `signalling_NaN`. These are methods of the `numeric_limits` type in the `limits` header; see the example below.

- *Nan* is only defined for floating point types: the test `has_quiet_NaN` is false for other types such as `bool` or `int`.
- Even though `complex` is built on top of floating point types, there is no *Nan* for it.

Code:	Output:
<pre>// limits/nan.cpp #include &lt;iostream&gt; #include &lt;limits&gt;  int main() {     std::cout &lt;&lt; "Double NaNs: {} {}\n";     std::cout &lt;&lt; std::numeric_limits&lt;double&gt;         ::quiet_NaN() &lt;&lt; " ";     std::cout &lt;&lt; std::numeric_limits&lt;double&gt;         ::signaling_NaN() &lt;&lt; "\n";     std::cout &lt;&lt; "zero divided by zero: {} {}\n";     std::cout &lt;&lt; 0 / 0.0 &lt;&lt; " ";     std::cout &lt;&lt; "Int has NaN: {} {}\n";     std::cout &lt;&lt; std::numeric_limits&lt;int&gt;         ::has_quiet_NaN() &lt;&lt; "\n"; }</pre>	<pre>[limits] nan: Double NaNs: nan nan zero divided by zero: nan Int has NaN: false</pre>

### 26.3.4.1 Tests

There are tests for detecting whether a number is *Inf*, *Nan*, et cetera. However, using these may slow a computation down.

The functions `isinf`, `isnan`, `isfinite`, `isnormal` are defined for the floating point types (`float`, `double`, `long double`), returning a `bool`.

```
#include <math.h>
isnan(-1.0/0.0); // false
isnan(sqrt(-1.0)); // true
isinf(-1.0/0.0); // true
isinf(sqrt(-1.0)); // false
```

### 26.3.5 Common numbers

```
#include <numbers>
static constexpr float pi = std::numbers::pi;
```

## 26.4 lvalue vs rvalue

The terms ‘lvalue’ and ‘rvalue’ sometimes appear in compiler error messages.

```

int foo() {return 2;}

int main()
{
    foo() = 2;

    return 0;
}

# gives:
test.c: In function 'main':
test.c:8:5: error: lvalue required as left operand of assignment

```

See the ‘lvalue’ and ‘left operand’? To first order of approximation you’re forgiven for thinking that an *lvalue* is something on the left side of an assignment. The name actually means ‘locator value’: something that’s associated with a specific location in memory. Thus an lvalue is, also loosely, something that can be modified.

An *rvalue* is then something that appears on the right side of an assignment, but is really defined as everything that’s not an lvalue. Typically, rvalues can not be modified.

The assignment `x=1` is legal because a variable `x` is at some specific location in memory, so it can be assigned to. On the other hand, `x+1=1` is not legal, since `x+1` is at best a temporary, therefore not at a specific memory location, and thus not an lvalue.

Less trivial examples:

```

int foo() { x = 1; return x; }
int main() {
    foo() = 2;
}

```

is not legal because `foo` does not return an lvalue. However,

```

class foo {
private:
    int x;
public:
    int &xfoo() { return x; };
};
int main() {
    foo x;
    x.xfoo() = 2;
}

```

is legal because the function `xfoo` returns a reference to the non-temporary variable `x` of the `foo` object.

Not every lvalue can be assigned to: in

```
const int a = 2;
```

the variable `a` is an lvalue, but can not appear on the left hand side of an assignment.

### 26.4.1 Conversion

Most lvalues can quickly be converted to rvalues:

```
int a = 1;
int b = a+1;
```

Here `a` first functions as lvalue, but becomes an rvalue in the second line.

The ampersand operator takes an lvalue and gives an rvalue:

```
int i;
int *a = &i;
&i = 5; // wrong
```

#### 26.4.2 References

The ampersand operator yields a reference. It needs to be assigned from an lvalue, so

```
std::string &s = std::string(); // wrong
```

is illegal. The type of `s` is an ‘lvalue reference’ and it can not be assigned from an rvalue.

On the other hand

```
const std::string &s = std::string();
```

works, since `s` can not be modified any further.

#### 26.4.3 Rvalue references

A new feature of C++ is intended to minimize the amount of data copying through *move semantics*.

Consider a copy assignment operator

```
BigThing& operator=( const BigThing &other ) {
    BigThing tmp(other); // standard copy
    std::swap( /*tmp data into my data */ );
    return *this;
};
```

This calls a copy constructor and a destructor on `tmp`. (The use of a temporary makes this safe under exceptions. The `swap` method never throws an exception, so there is no danger of half-copied memory.)

However, if you assign

```
thing = BigThing(stuff);
```

Now a constructor and destructor is called for the temporary rvalue object on the right-hand side.

Using a syntax that is new in C++, we create an *rvalue reference*:

```
BigThing& operator=( BigThing &&other ) {
    swap( /* other into me */ );
    return *this;
}
```

## 26.5 Move semantics

With an overloaded operator, such as addition, on matrices (or any other big object):

```
Matrix operator+(Matrix &a, Matrix &b);
```

the actual addition will involve a copy:

```
Matrix c = a+b;
```

Use a move constructor:

```
class Matrix {
private:
    Representation rep;
public:
    Matrix(Matrix &&a) {
        rep = a.rep;
        a.rep = {};
    }
};
```

## 26.6 Graphics

C++ has no built-in graphics facilities, so you have to use external libraries such as *OpenFrameworks*, <https://openframeworks.cc>.

## 26.7 Standards timeline

Each standard has many changes over the previous.

If you want to detect what language standard you are compiling with, use the `__cplusplus` macro:

<b>Code:</b>	<b>Output:</b>
<pre>// basic/version.cpp cout &lt;&lt; "C++ version: " &lt;&lt; __cplusplus &lt;&lt; '\n';</pre>	<pre>[basic] version: C++ version: 201703</pre>

This returns a `long int` with possible values 199711, 201103, 201402, 201703, 202002.

Here are some of the highlights of the various standards.

### 26.7.1 C++98/C++03

Of the C++03 standard we only highlight deprecated features.

- `auto_ptr` was an early attempt at smart pointers. It is deprecated, and C++17 compilers will actually issue an error on it. For current smart pointers see chapter 16.

### 26.7.2 C++11

- `auto`

```
const auto count = std::count
    (begin(vec), end(vec), value);
```

The `count` variable now gets the type of whatever `vec` contained.

- Range-based for. We have been treating this as the base case, for instance in section 10.2. The pre-C++11 mechanism, using an *iterator* (section 14.2.1) is largely obviated.
- Lambdas. See chapter 13.
- Chrono; see section 24.8.
- Variadic templates.
- Smart pointers; see chapter 16.

```
unique_ptr<int> iptr( new int(5) );
```

This fixes problems with `auto_ptr`.

- `constexpr`

```
constexpr int get_value() {
    return 5*3;
}
```

### 26.7.3 C++14

C++14 can be considered a bug fix on C++11. It simplifies a number of things and makes them more elegant.

- Auto return type deduction:

```
auto f() {
    SomeType something;
    return something;
}
```

- Generic lambdas (section 13.3.2)

```
const auto count = std::count(begin(vec), end(vec),
    [] ( const auto i ) { return i<3; }
);
```

Also more sophisticated capture expressions.

- `constexpr`

```
constexpr int get_value() {
    int val = 5;
    int val2 = 3;
    return val*val2
}
```

### 26.7.4 C++17

- Optional; section 24.6.2.
- Structured binding declarations as an easier way of dissecting tuples; section 24.5.
- Init statement in conditionals; section 5.5.3.

### 26.7.5 C++20

First four big additions:

- *modules*: these offer a better interface specification than using *header files*; see section 19.5.
- *coroutines*, another form of parallelism.
- *concepts* including in the standard library via ranges; section 22.4.
- *ranges*

Core language improvements:

- *spaceship operator* including in the standard library
- broad use of normal C++ for direct compile-time programming, without resorting to template meta programming (see last trip reports)
- Designated initialization for data members.
- *consteval* and *constinit*

Library additions:

- *calendars* and *time zones*
- *text formatting*
- *span*. See section 10.8.3.
- *numbers*. Section 26.3.5.
- Safe integer/unsigned comparison; section 26.3.2.2; integers are guaranteed two's complement.

Finally, various concurrency improvements.

Here is a summary with examples: <https://oleksandrkvli.github.io/2021/04/02/cpp-20-overview.html>. Another good lecture: <https://www.youtube.com/watch?v=BhMUdb5fRcE>

### 26.7.6 C++23

- *mdspan* offers multi-dimensional array; this is an extension of the C++17 *span* mechanism.



## Chapter 27

### Graphics

The C++ language and standard library do not have graphics components. However, the following projects exist.

#### 27.1 SFML

<https://www.sfml-dev.org/>

#### 27.2 VT100 cursor control

Often you want some sort of graphical or animated output. However, not all programming languages generate visual output equally easily. There are very powerful video/graphics libraries in C++, but these are also non-trivial to use. There is a simpler way out.

For simple output you can make some low-budget graphics. Every *terminal emulator* under the sun supports *VT100 cursor control* codes [27], named after the 1980s *DEC VT100* terminal. With these you can send certain magic sequences to your screen to control cursor positioning, or set other terminal properties such as switch to alternate character sets.

While a large number of control codes are available, the following minimum often suffices.

- Wipe the screen clean by ‘printing’:

```
#include <cstdio>
printf( "%c[2J", (char)27 );
```

- Then position the cursor at the top left coordinate:

```
printf( "%c[0;0H", (char)27 );
```

With the `format` header you could also:

```
cout << format("\u001b[2J");
cout << format("\u001b[0;0H");
// or use std::print
```



## Chapter 28

### C++ for C programmers

#### 28.1 I/O

Functions such as `printf` and `scanf` can be dispensed with. Instead, use `cout` (and `cerr`) and `cin`. Even more modern are the `format` and `print` libraries of C++20 and C++23.

Chapter 12.

#### 28.2 Arrays

Arrays through square bracket notation are unsafe. They are basically a pointer, which means they carry no information beyond the memory location.

It is much better to use `vector`. Use range-based loops, even if you use bracket notation.

Chapter 10.

Vectors own their data exclusively, so having multiple C style pointers into the same data act like so many arrays does not work. For that, use the `span`; section 10.8.3.

##### 28.2.1 Vectors from C arrays

Suppose you have to interface to a C code that uses `malloc`. Vectors have advantages, such as that they know their size, you may want to wrap these C style arrays in a vector object. This can be done using a *range constructor*:

```
vector<double> x( pointer_to_first, pointer_after_last );
```

Such vectors can still be used dynamically, but this may give a memory leak and other possibly unwanted behavior:

```
Code:  
// array/cvector.cpp  
float *x;  
x = (float*)malloc(length*sizeof(float))  
;  
/* ... */  
vector<float> xvector(x, x+length);  
cout << "xvector has size: " << xvector.  
size() << '\n';  
xvector.push_back(5);  
cout  
    << "Push back was successful" << '\n';  
cout << "pushed element: "  
    << xvector.at(length) << '\n';  
cout << "original array: "  
    << x[length] << '\n';
```

```
Output:  
[array] cvector:  
xvector has size: 53  
Push back was successful  
pushed element: 5  
original array: 0
```

### 28.3 Dynamic storage

Another advantage of vectors and other containers is the *RAII* mechanism, which implies that dynamic storage automatically gets deallocated when leaving a scope. Section 10.9.5. (For safe dynamic storage that transcends scope, see smart pointers discussed below.)

RAII stands for ‘Resource Allocation Is Initialization’. This means that it is no longer possible to write

```
double *x;  
if (something1) x = malloc(10);  
if (something2) x[0];
```

which may give a memory error. Instead, declaration of the name and allocation of the storage are one indivisible action.

On the other hand:

```
vector<double> x(10);
```

declares the variable `x`, allocates the dynamic storage, and initializes it.

### 28.4 Strings

A C *string* is a character array with a *null terminator*. On the other hand, a `string` is an object with operations defined on it.

Chapter 11.

## 28.5 Pointers

Many of the uses for *C pointers*, which are really addresses, have gone away.

- Strings are done through `std::string`, not character arrays; see above.
- Arrays can largely be done through `std::vector`, rather than `malloc`; see above.
- Traversing arrays and vectors can be done with ranges; section 10.2.
- To pass an argument *by reference*, use a *reference*. Section 7.5.
- Anything that obeys a scope should be created through a *constructor*, rather than using `malloc`.

There are some legitimate needs for pointers, such as Objects on the heap. In that case, use `shared_ptr` or `unique_ptr`; section 16.1. The C pointers are now called *bare pointers*, and they can still be used for ‘non-owning’ occurrences of pointers.

### 28.5.1 Parameter passing

No longer by address: now true references! Section 7.5.

### 28.5.2 Addresses

C programmers are accustomed to using the ampersand for getting the address of variables, and in C++: objects. C++ programmers need to beware that it’s possible to redefine the ampersand:

```
T operator& () { /* stuff */ };
```

If you absolutely need an address, use the `addressof` function.

## 28.6 Objects

Objects are structures with functions attached to them. Chapter 9.

## 28.7 Namespaces

No longer name conflicts from loading two packages: each can have its own namespace. Chapter 20.

## 28.8 Templates

If you find yourself writing the same function for a number of types, you’ll love templates. Chapter 22.

## 28.9 Obscure stuff

### 28.9.1 Lambda

Function expressions. Chapter 13.

### 28.9.2 Const

Functions and arguments can be declared const. This helps the compiler. Section 18.1.

### 28.9.3 Lvalue and rvalue

Section 26.4.

## Chapter 29

### C++ review questions

#### 29.1 Arithmetic

1. Given

```
int n;
```

write code that uses elementary mathematical operators to compute n-cubed:  $n^3$ .

Do you get the correct result for all  $n$ ? Explain.

2. What is the output of:

```
int m=32, n=17;
cout << n%m << endl;
```

#### 29.2 Looping

1. Suppose a function

```
bool f(int);
```

is given, which is true for some positive input value. Write a main program that finds the smallest positive input value for which  $f$  is true.

2. Suppose a function

```
bool f(int);
```

is given, which is true for some negative input value. Write a main program that finds the (negative) input with smallest absolute value for which  $f$  is true.

#### 29.3 Functions

**Exercise 29.1.** The following code snippet computes in a loop the recurrence

$$v_{i+1} = av_i + b, \quad v_0 \text{ given.}$$

Write a recursive function

```
float v = value_n(n, a, b, v0);
```

that computes the value  $v_n$  for  $n \geq 0$ .

## 29.4 Vectors

**Exercise 29.2.** The following program has several syntax and logical errors. The intended purpose is to read an integer  $N$ , and sort the integers  $1, \dots, N$  into two vectors, one for the odds and one for the evens. The odds should then be multiplied by two.

Your assignment is to debug this program. For 10 points of credit, find 10 errors and correct them. Extra errors found will count as bonus points. For logic errors, that is, places that are syntactically correct, but still ‘do the wrong thing’, indicate in a few words the problem with the program logic.

```
#include <iostream>
using std::cout; using std::cin;
using std::vector;

int main() {
    vector<int> evens, odd;
    cout << "Enter an integer value " << endl;
    cin << N;
    for (i=0; i<N; i++) {
        if (i%2==0) {
            odds.push_back(i);
        } else
            evens.push_back(i);
    }
    for ( auto o : odds )
        o /= 2
    return 1
}
```

## 29.5 Vectors

**Exercise 29.3.** Take another look at exercise 29.1. Now assume that you want to save the values  $v_i$  in an array `vector<float> values`. Write code that does that, using first the iterative, then the recursive computation. Which do you prefer?

## 29.6 Objects

**Exercise 29.4.** Let a class `Point` class be given. How would you design a class `SetOfPoints` (which models a set of points) so that you could write

```
Point p1, p2, p3;
SetOfPoints pointset;
// add points to the set:
pointset.add(p1); pointset.add(p2);
```

Give the relevant data members and methods of the class.

**Exercise 29.5.** You are programming a video game. There are moving elements, and you want to have an object for each. Moving elements need to have a method `move` with an argument that indicates a time duration, and this method updates the position of the element, using the speed of that object and the duration.

Supply the missing bits of code.

```
class position {
    /* ... */
public:
    position() {};
    position(int initial) { /* ... */ };
    void move(int distance) { /* ... */ };
};

class actor {
protected:
    int speed;
    position current;

public:
    actor() { current = position(0); };
    void move(int duration) {
        /* THIS IS THE EXERCISE: */
        /* write the body of this function */
    };
};

class human : public actor {
public:
    human() // EXERCISE: write the constructor
};

class airplane : public actor {
public:
    airplane() // EXERCISE: write the constructor
};

int main() {
    human Alice;
    airplane Seven47;
    Alice.move( 5 );
    Seven47.move( 5 );
}
```



## **PART III**

### **FORTRAN**



## Chapter 30

### Basics of Fortran

Fortran is an old programming language, dating back to the 1950s, and the first ‘high level programming language’ that was widely used. In a way, the fields of programming language design and compiler writing started with Fortran, rather than this language being based on established fields. Thus, the design of Fortran has some idiosyncrasies that later designed languages have not adopted. Many of these are now ‘deprecated’ or simply inadvisable. Fortunately, it is possible to write Fortran in a way that is every bit as modern and sophisticated as other current languages.

In this part of the book, you will learn safe practices for writing Fortran. Occasionally we will not mention practices that you will come across in old Fortran codes, but that we would not advise you taking up. While this exposition of Fortran can stand on its own, we will in places point out explicitly differences with C++.

#### 30.1 Source format

Fortran started in the era when programs were stored on *punch cards*. Those had 80 columns, so a line of Fortran source code could not have more than 80 characters. Also, the first 6 characters had special meaning. This is referred to as *fixed format*. However, starting with *Fortran 90* it became possible to have *free format*, which allowed longer lines without special meaning for the initial columns.

There are further differences between the two formats (notably continuation lines) but we will only discuss free format in this course.

Many compilers have a convention for indicating the source format by the file name extension:

- `f` and `F` are the extensions for old-style fixed format; and
- `f90` and `F90` are the extensions for new free format.

Capital letters indicate that the *C preprocessor* is applied to the file. For this course we will use the `F90` extension.

#### 30.2 Compiling Fortran

The minimal Fortran program is:

```
// basicf/emptyprog.F90
Program SomeProgram
    ! stuff goes here
End Program SomeProgram
```

You would compile this:

```
yourfortrancompiler -o myprogram myprogram.F90
```

and then execute with

```
./myprogram
```

For Fortran programs, the compiler is *gfortran* for the GNU compiler, and *ifort* for Intel.

**Exercise 30.1.** Add the line

```
print *, "Hello world!"
```

to the empty program, and compile and run it.

Fortran ignores case in both keywords and identifiers. Keywords such as `Program` in the above program can thus just as well be written as `PrOgRaM`.

A program optionally has a `stop` statement, which can return a message to the OS.

**Code:**

```
// basicf/stop.F90
Program SomeProgram
    stop 'the code stops here'
End Program SomeProgram
```

**Output:**  
**[basicf] stop:**  
*STOP the code stops here*

Additionally, a numeric code returned by `stop`

```
stop 1
```

can be queried with the `$?` shell parameter:

**Code:**

```
// basicf/stopreturn.F90
Program SomeProgram
    stop 17
End Program SomeProgram
```

**Output:**  
**[basicf] return:**  
*./stopreturn || code=\$? \&& echo Return code*  
*→is \$code*  
*STOP 17*  
*Return code is 17*

### 30.3 Main program

Fortran does not use curly brackets to delineate blocks, instead you will find `end` statements. The very first one appears right when you start writing your program: a Fortran program needs to start with a `Program` line, and end with `End Program`. The program needs to have a name on both lines:

```
// basicf/emptyprog.F90
Program SomeProgram
    ! stuff goes here
End Program SomeProgram
```

and you can not use that name for any entities in the program.

**Remark** The emacs editor will supply the block type and name if you supply the ‘end’ and hit the TAB or RETURN key; see section 2.1.1.

#### 30.3.1 Program structure

Unlike C++, Fortran can not mix variable declarations and executable statements, so both the main program and any subprograms have roughly a structure:

```
Program foo
< declarations >
< statements >
End Program foo
```

Another thing to note is that there are no include directives. Fortran does not have a ‘standard library’ such as C++ that needs to be explicitly included. Or you could say that the Fortran standard library is always by default included.

#### 30.3.2 Statements

Let’s say a word about layout. Fortran has a ‘one line, one statement’ principle, stemming from its *punch card* days.

- As long as a statement fits on one line, you don’t have to terminate it explicitly with something like a semicolon:

```
x = 1
y = 2
```

- If you want to put two statements on one line, you have to terminate the first one:

```
x = 1; y = 2
```

But watch out for the line length: this is often limited to 132 characters.

- If a statement spans more than one line, all but the first line need to have an explicit *continuation character*, the ampersand:

```
x = very &
long &
expression
```

### 30.3.3 Comments

Fortran knows only single-line *comments*, indicated by an exclamation point:

```
x = 1 ! set x to one
```

Everything from the exclamation point to the end of the line is ignored.

Maybe not entirely obvious: you can have a comment after a continuation character:

```
x = f(a) & ! term1
+ g(b)      ! term2
```

**Remark** In Fortran77, 19 continuation lines were allowed. In Fortran95 this number was 40. As of the Fortran2003 standard, a line can be continued 256 times.

## 30.4 Variables

Unlike in C++, where you can declare a variable right before you need it, Fortran wants its variables declared near the top of the program or subprogram:

```
Program YourProgram
  implicit none
  ! variable declaration
  ! executable code
End Program YourProgram
```

The `implicit none` should always be included; see section 30.4.1.1 for an explanation.

A variable declaration looks like:

```
type [ , attributes ] :: name1 [ , name2, .... ]
```

where

- we use the common grammar shorthand that [ something ] stands for an optional ‘something’;
- *type* is most commonly `integer`, `real(4)`, `real(8)`, `logical`. See below; section 30.4.1.
- the optional *attributes* are things such as `dimension`, `allocatable`, `intent`, `parameter` et cetera.
- *name* is something you come up with. This has to start with a letter. Unusually, variable names are case-insensitive. Thus,

```
Integer :: MYVAR
MyVar = 2
print *,myvar
```

is perfectly legal.

**Remark** In Fortran66 there was a limit of six characters to the length of a variable name, though many compilers had extensions to this. As of the Fortran2003 standard, a variable name can be 63 characters long. The built-in data types of Fortran:

- Numeric: `Integer`, `Real`, `Complex`
- precision control:

```
Integer :: i
Integer(4) :: i4
Integer(8) :: i8
```

This usually corresponds to number of bytes; see textbook for full story.

- Logical: `Logical`.
- Character: `Character`. Strings are realized as arrays of characters.
- Derived types (like C++ structures or classes): `Type`

Some variables are not intended ever to change, such as if you introduce a variable `pi` with value 3.14159. You can mark this name as being a synonym for the value, rather than a variable you can assign to, with the `parameter` keyword.

```
real,parameter :: pi = 3.141592
```

In chapter 39 you will see that `parameters` are often used for defining the size of an array.

Further specifications for numerical precision are discussed in section 30.4.1.2. Strings are discussed in chapter 35.

### 30.4.1 Declarations

#### 30.4.1.1 Implicit declarations

Fortran has a somewhat unusual treatment of variable types: if you don't specify what data type a variable is, Fortran will deduce it from a simple rule, based on the first character of the name. This is a very dangerous practice, so we advocate putting a line

```
implicit none
```

immediately after any program or subprogram header. Now every variable needs to be given a type explicitly in a declaration.

#### 30.4.1.2 Variable 'kind's

Fortran has several mechanisms for indicating the precision of a numerical type.

```
// ftype/storage.F90
integer(2) :: i2
integer(4) :: i4
integer(8) :: i8

real(4) :: r4
real(8) :: r8
real(16) :: r16

complex(8) :: c8
complex(16) :: c16
```

```
complex*32 :: c32
```

This often corresponds to the number of bytes used, **but not always**. It is technically a numerical *kind selector*, and it is nothing more than an identifier for a specific type.

### 30.4.2 Initialization

Variables can be initialized in their declaration:

```
integer :: i=2
real(4) :: x = 1.5
```

That this is done at compile time, leading to a common error:

```
subroutine foo()
  implicit none
  integer :: i=2
  print *,i
  i = 3
end subroutine foo
```

On the first subroutine call *i* is printed with its initialized value, but on the second call this initialization is not repeated, and the previous value of 3 is remembered.

## 30.5 Complex numbers

A complex number is a pair of real numbers. Complex constants can be written with a parenthesis notation, but to form a complex number from two real variables requires the **CMPLEX** function. You can also use this function to force a real number to complex, so that subsequent computations are done in the complex realm.

Real and imaginary parts can be extracted with the function **real** and **aimag**.

Complex constants are written as a pair of reals in parentheses.

There are some basic operations.

**Code:**

```
// basicf/roots.F90
Complex :: &
  fourtyfivedegrees = (1.,1.), &
  number,rotated
Real :: x,y
print *, "45 degrees:", fourtyfivedegrees
x = 3. ; y = 1.; number = cmplx(x,y)
rotated = number * fourtyfivedegrees
print '("Rotated number has Re=",f5.2," Im=",
      f5.2)',&
      real(rotated), aimag(rotated)
```

**Output:**

```
[basicf] complex:
45 degrees:
  ↪(1.00000000,1.00000000)
Rotated number has Re= 2.00 Im=
  ↪4.00
```

The imaginary root *i* is not predefined. Use

```
Complex,parameter :: i = (0,1)
```

In Fortran2008, `Complex` is a derived type, and the real/imaginary parts can be extracted as

```
print *,rotated%re,rotated%im
```

**Code:**

```
// basicf/complexf08.F90
print *, "45 degrees:", fourtyfivedegrees
x = 3. ; y = 1.; number = cmplx(x,y)
rotated = number * fourtyfivedegrees
print'("Rotated number has Re=",f5.2," Im=",
      f5.2)',&
      rotated%re,rotated%im
```

**Output:**

```
[basicf] complexf08:
45 degrees:
  ↪(1.00000000,1.00000000)
Rotated number has Re= 2.00
  ↪Im= 4.00
```

**Exercise 30.2.** Write a program to compute the complex roots of the quadratic equation

$$ax^2 + bx + c = 0$$

The variables `a`, `b`, `c` have to be real, but use the `cmplx` function to force the computation of the roots to happen in the complex domain.

## 30.6 Expressions

Fortran has arithmetic, logical, and string expressions.

- Arithmetic expressions look more or less the way you would expect them to. The only unusual operator is the power operator: `x**.5` is the square root of variable `x`.
- For boolean expressions there are constants `.true.` and `.false.`; operators are likewise enclosed in dots: `.and.` and such. Boolean variables are of type `Logical`. See section 30.19 for more.
- String handling will be discussed in chapter 35.

## 30.7 Bit operations

As of Fortran95 there are functions for bitwise operations:

- `btest(word, pos)` returns a `logical` if bit `pos` in `word` is set.
- `ibits(word, pos, len)` returns an integer (of the same kind as `word`) with bits  $p, \dots, p + \ell - 1$  (extending leftward) right-adjusted.
- `ibset(i, pos)` takes an integer and returns the integer resulting from setting bit `pos` to 1. Likewise, `ibclr` clears that bit.
- `iand`, `ior`, `ieor` all operate on two integers, returning the bitwise and/or/xor result.
- `mvbits(from, frompos, len, to, tpos)` copies a range of bits between two integers.

## 30.8 Commandline arguments

Modern Fortran has functions for querying *commandline arguments*. First of all `command_argument_count` queries the number of arguments. This does not include the command itself, so this is one less than the C/C++ `argc` argument to main.

```
// basicf/command.F90
if (command_argument_count() == 0) then
    print *, "This program needs an argument"
    stop 1
end if
```

The command can be retrieved with `get_command`.

The commandline arguments are retrieved with `get_command_argument`. These are strings as in C/C++, but you have to specify their length in advance:

```
// basicf/command.F90
character(len=10) :: size_string
integer :: size_num
```

Converting this string to an integer or so takes a little format trickery:

```
// basicf/command.F90
call get_command_argument(number=1,value=size_string)
read(size_string,'(i3)') size_num
```

(see section 41.3.)

## 30.9 Fortran type kinds

### 30.9.1 Kind selection

Kinds can be used to ask for a type with specified precision.

- For integers you can specify the number of decimal digits with `selected_int_kind($n$)`.
- For floating point numbers can specify the number of significant digits, and optionally the decimal exponent range with `selected_real_kind($p$, $r$)` of significant digits.

Conversely, the properties of such types can be retrieved again with the functions `precision` (not for integers), `range`, `storage_size`.

Declaration of precision and/or range:

```
Code:
// basicf/kind.F90
integer,parameter :: &
    i12 = selected_int_kind(12), &
    p6 = selected_real_kind(6), &
    p10r100 = selected_real_kind(10,100), &
    r400 = selected_real_kind(r=400), &
    p20 = selected_real_kind(20), &
    p40 = selected_real_kind(40)
integer(kind=i12) :: i
real(kind=p6) :: x
real(kind=p10r100) :: y
real(kind=r400) :: z
real(kind=p20) :: p
```

```
Output:
[basicf] kind:
Kinds:   8      4      8     16
        ↪16     -1
Precision / range / bits:
integer 12 :    0    18    64
precision 6:    6    37    32
p=10 r=100 :   15   307   64
range=400 :   33 4931 128
p=20       :   33 4931 128
```

Likewise, you can specify the precision of a constant. Writing `3.14` will usually be a single precision real.

Adding single/double precision constants, print as double:

```
Code:
// basicf/e0.F90
real(8) :: x,y,z
x = 1.
y = .1
z = x+y
print *, z
x = 1.d0
y = .1d0
z = x+y
print *, z
```

```
Output:
[basicf] e0:
1.100000014901161
1.1000000000000001
```

You can query how many bytes a data type takes with `kind`.

Number of bytes determines numerical precision:

- Computations in 4-byte have relative error  $\approx 10^{-6}$
- Computations in 8-byte have relative error  $\approx 10^{-15}$

Also different exponent range: max  $10^{\pm 50}$  and  $10^{\pm 300}$  respectively.

F08: `storage_size` reports number of bytes.

F95: `bit_size` works on integers only.

`c_sizeof` reports number of bytes, requires `iso_c_binding` module.

**Code:**

```
// basicf/binding.F90
use iso_c_binding
implicit none
integer,parameter :: &
    p6 = selected_real_kind(6), &
    p12 = selected_real_kind(12)
real(kind=p6) :: x4
real(kind=p12) :: x8
10 format(i2" digits takes",i3," bytes")
print 10,6,c_sizeof(x4)
print 10,12,c_sizeof(x8)
```

**Output:**

```
[basicf] binding:
 6 digits takes 4 bytes
12 digits takes 8 bytes
```

Force a constant to be `real(8)`:

```
real(8) :: x,y
x = 3.14d0
y = 6.022e-23
```

- Use a compiler flag such as `-r8` to force all reals to be 8-byte.
- Write `3.14d0`
- `x = real(3.14, kind=8)`

### 30.9.2 Range

You can use the function `huge` to query the maximum value of a type.

**Code:**

```
// typedefs.F90
Integer :: idef
Real :: rdef
Real(8) :: rdouble

print 10,"integer is kind",kind(idef)
print 10,"integer max is",huge(idef)
print 10,"real is kind",kind(rdef)
print 15,"real max is",huge(rdef)
print 10,"real8 is kind",kind(rdouble)
print 15,"real8 max is",huge(rdouble)
```

**Output:**

```
[typedef] def:
integer is kind          4
integer max is 2147483647
real is kind            4
real max is 0.3403E+39
real8 is kind           8
real8 max is 0.1798+309
```

With *ISO bindings* there is a more systematic approach.

Integers:

```
// typedef/inttypes.F90
Integer(kind=Int8) :: i8
Integer(kind=Int16) :: i16
Integer(kind=Int32) :: i32
Integer(kind=Int64) :: i64
```

**Code:**

```
// typef/inttypes.F90
print 10,"Checking on supported types:"
print 10,"number of defined int types:",size
  (INTEGER_KINDS)
print 10,"these are the supported types:",
  INTEGER_KINDS
print 15,"Pre-defined types INT8,INT16,INT32
,INT64:",&
  INT8,INT16,INT32,INT64
print *
print 20,"kind Int8 max is",huge(i8)
print 20,"kind Int16 max is",huge(i16)
print 20,"kind Int32 max is",huge(i32)
print 20,"kind Int64 max is",huge(i64)
```

**Output:**

```
[typef] int:
Checking on supported
  ↳types:
  number of defined int
  ↳types: 5
these are the supported
  ↳types: 1 2 4 8
  16
Pre-defined types
  ↳INT8, INT16, INT32, INT64
  ↳ 1 2 4 8

kind Int8 max is
  ↳ 127
kind Int16 max is
  ↳ 32767
kind Int32 max is
  ↳ 2147483647
kind Int64 max is
  ↳9223372036854775807
```

Floating point numbers:

```
// basicf/iso.F90
use iso_fortran_env
implicit none
real(kind=real32) :: x32
real(kind=real64) :: x64
print *, "32 bit max float:",huge(x32)
print *, "64 bit max float:",huge(x64)
```

## 30.10 Quick comparison Fortran vs C++

### 30.10.1 Statements

Some of it is much like C++:

- Assignments:

```
x = y
x = 2*y / (a+b)
z1 = 5; z2 = 6
```

(Note the lack of semicolons at the end of statements.)

- I/O
- conditionals and loops

Different:

- function definition and calls
- array syntax
- object oriented programming
- modules

### 30.10.2 Input/Output, or I/O as we say

- Input:

```
READ *, n
```

- Output:

```
PRINT *, n
```

There is also `Write`.

The ‘star’ indicates that default formatting is used.

Other syntax for read/write with files and formats.

### 30.10.3 Expressions

- Pretty much as in C++
- Exception: `r**a` for power  $r^a$ .
- Modulus (the % operator in C++) is a function: `MOD(7,3)`.

- Long form:  
.and. .not. .or.  
.lt. .le. .eq. .ne. .ge. .gt.  
.true. .false.
- Short form:  
< <= == /= > >=

Conversion is done through functions.

- `INT`: truncation; `NINT` rounding
- `REAL`, `FLOAT`, `SNGL`, `DBLE`
- `CMPLX`, `CONJG`, `AIMAG`

<http://userweb.eng.gla.ac.uk/peter.smart/com/com/f77-conv.htm>

Complex numbers exist; section 30.5.

Strings are delimited by single or double quotes.

For more, see chapter 35.

## 30.11 Review questions

**Exercise 30.3.** What is the output for this fragment, assuming *i*, *j* are integers?

```
// basicf/div.F90
integer :: idiv
/* ... */
i = 3 ; j = 2 ; idiv = i/j
print *,idiv
```

**Exercise 30.4.** What is the output for this fragment, assuming *i*, *j* are integers?

```
// basicf/div.F90
real    :: fdiv
/* ... */
i = 3 ; j = 2 ; fdiv = i/j
print *,fdiv
```

**Exercise 30.5.** In declarations

```
real(4) :: x
real(8) :: y
```

what do the 4 and 8 stand for?

What is the practical implication of using the one or the other?

**Exercise 30.6.** Write a program that :

- displays the message Type a number,
- accepts an integer number from you (use Read),
- makes another variable that is three times that integer plus one,
- and then prints out the second variable.

**Exercise 30.7.** In the following code, if *value* is nonzero, what do expect about the output?

```
// basicf/d0.F90
real(8) :: value8,should_be_value
real(4) :: value4
/* ... */
print *,.." original value was:",value8
value4 = value8
print *,.." copied to single:",value4
should_be_value = value4
print *,.." copied back to double:",should_be_value
print *,.."Difference:",value8-should_be_value
```



# Chapter 31

## Conditionals

### 31.1 Forms of the conditional statement

The Fortran conditional statement uses the `if` keyword:

Single line conditional:

```
if ( test ) statement
```

The full if-statement is:

```
if ( something ) then
  !! something_doing
else
  !! otherwise_else
end if
```

The ‘else’ part is optional; you can nest conditionals.

You can label conditionals, which is good for readability but adds no functionality:

```
checkx: if ( ... some test on x ... ) then
checky:   if ( ... some test on y ... ) then
            ... code ...
        end if checky
    else checkx
        ... code ...
    end if checkx
```

### 31.2 Operators

## 31. Conditionals

---

Operator	old style	meaning	example
<code>==</code>	<code>.eq.</code>	equals	<code>x==y-1</code>
<code>/=</code>	<code>.ne.</code>	not equals	<code>x*x/=5</code>
<code>&gt;</code>	<code>.gt.</code>	greater	<code>y&gt;x-1</code>
<code>&gt;=</code>	<code>.ge.</code>	greater or equal	<code>sqrt(y)&gt;=7</code>
<code>&lt;</code>	<code>.lt.</code>	less than	
<code>&lt;=</code>	<code>.le.</code>	less or equal	
	<code>.and. .or.</code>	and, or	<code>x&lt;1 .and. x&gt;0</code>
	<code>.not.</code>	not	<code>.not.( x&gt;1 .and. x&lt;2 )</code>
	<code>.eqv.</code>	equiv (iff, not XOR)	
	<code>.neqv.</code>	not equiv (XOR)	

The logical operators such as `.AND.` are not short-cut as in C++. Clauses can be evaluated in any order.

**Exercise 31.1.** Read in three grades: Algebra, Biology, Chemistry, each on a scale 1 ··· 10. Compute the average grade, with the conditions:

- Algebra is always included.
- Biology is only included if it increases the average.
- Chemistry is only included if it is 6 or more.

### 31.3 Select statement

The Fortran equivalent of the C++ `case` statement is `select`. It takes single values or ranges; works for integers and character strings.

Test single values or ranges, integers or characters:

```
// basicf/select.F90
Select Case (i)
Case (: -1) ! range one and less
    print *, "Negative"
Case (5)
    print *, "Five!"
Case (0)
    print *, "Zero."
Case (1:4,6:) ! other cases, can not have (1:)
    print *, "Positive"
end Select
```

Compiler does checking on overlapping cases!

Case values need to be constant expressions.

The default case is covered with a `case default` case.

### 31.4 Boolean variables

The Fortran type for booleans is `Logical`.

The two literals are `.true.` and `.false.`

**Exercise 31.2.** Print a boolean variable. What does the output look like in the true and false case?

## 31.5      Obsolete conditionals

Old versions of Fortran had other forms of the `if` statement, which you may still encounter in codes. The `if, arithmetic` was declared obsolescent in Fortran90 and was deleted in Fortran2018.

## 31.6      Review questions

**Exercise 31.3.** What is a conceptual difference between the C++ `switch` and the Fortran `Select` statement?



# Chapter 32

## Loop constructs

### 32.1 Loop types

Fortran has the usual indexed and ‘while’ loops. There are variants of the basic loop, and both use the `do` keyword. The simplest loop has a loop variable, an upper bound, and a lower bound.

```
integer :: i  
  
do i=1,10  
    ! code with i  
end do
```

You can include a step size (which can be negative) as a third parameter:

By steps of 3:

```
do i=1,10,3  
    ! code with i  
end do
```

Counting down:

```
do i=10,1,-1  
    ! code with i  
end do
```

The loop variable is defined outside the loop, so it will have a value after the loop terminates.

- Fortran loops determine the iteration count before execution; a loop will run that many iterations, unless you `Exit`.
- You are not allowed to alter the iteration variable.
- Non-integer loop variables used to be allowed, no longer.

The while loop has a pre-test:

```
do while (i<1000)  
    print *,i  
    i = i*2  
end do
```

## 32.2 Interruptions of the control flow

For indeterminate looping, you can use the `while` test, or leave out the loop parameter altogether. In that case you need the `exit` statement to stop the iteration.

Loop without counter or while test:

```
do
    call random_number(x)
    if (x>.9) exit
    print *, "Nine out of ten exes agree"
end do
```

Compare to `break` in C++.

Skip rest of current iteration:

```
do i=1,100
    if (isprime(i)) cycle
    ! do something with non-prime
end do
```

Compare to `continue` in C++.

You can label loops

useful with `exit` statement:

```
// loopf/labeled.F90
outer: do i=1,10
    inner: do j=1,10
        test: if (i*j>42) then
            print *,i,j
            exit outer
        end if test
    end do inner
end do outer
```

The label needs to be on the same line as the `do`, and if you use a label, you need to mention it on the `end do` line.

Cycle and exit can apply to multiple levels, if the do-statements are labeled.

```
outer : do i = 1,10
inner : do j = 1,10
    if (i+j>15) exit outer
    if (i==j) cycle inner
end do inner
end do outer
```

### 32.3 Implied do-loops

There are do loops that you can write in a single line by an expression and a loop header. In effect, such an *implied do loop* becomes the sum of the indexed expressions. This is useful for I/O. For instance, iterate a simple expression:

If you loop over a print statement, each print statement is on a new line; use an implied loop to print on one line.

```
Print *, (2*i, i=1, 20)
```

You can iterate multiple expressions:

```
Print *, (2*i, 2*i+1, i=1, 20)
```

These loops can be nested:

```
Print *, ( (i*j, i=1, 20), j=1, 20 )
```

Also useful for [Read](#).

This construct is especially useful for printing arrays.

**Exercise 32.1.** Use the implied do-loop mechanism to print a triangle:

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

up to a number that is input.

### 32.4 Obsolete loop statements

Old versions of Fortran had other forms of the `do` statement, which you may still encounter in codes. As of Fortran2018, `do` loops have to end in `end do` or `continue`. Shared termination is likewise a deleted feature.

Fortran has a `goto` statement. While this was needed in the 1950 and 60s, nowadays it is considered bad programming practice. Most of its traditional uses can be covered with the `cycle` and `exit` statements. The `continue` statement, usually used as the target of a `goto`, is similarly rarely used anymore.

### 32.5 Review questions

**Exercise 32.2.** What is the output of:

```
do i=1,11,3
  print *, i
end do
```

What is the output of:

```
do i=1,3,11  
    print *,i  
end do
```

# Chapter 33

## Procedures

Programs can have subprograms: parts of code that for some reason you want to separate from the main program. The term for these is *procedure*. While this is actually a keyword, you will not see it until section 37.5; in this chapter we consider only `subroutine` and `function`.

If you structure your code in a single file, this is the recommended structure:

Simplest way of defining procedures:  
in `Contains` part of main program.

```
Program foo
  < declarations>
  < executable statements >
  Contains
    < procedure definitions >
End Program foo
```

Two types of procedures: functions and subroutines. More later.

That is, procedures are placed after the main program statements, separated by a `Contains` clause.

In general, these are the placements of procedures:

- Internal: after the `Contains` clause of a program:

```
Program foo
  ... stuff ...
  Contains
    Subroutine bar()
    End Subroutine bar
  End Program foo
```

This is the mode that we focus on in this chapter.

- In a `Module`; see section 37.2.
- Externally: the procedure is not internal to a `Program` or `Module`. This can happen in the case of 3rd party libraries, or code linked in from another language. In this case it's safest to declare the procedure through an `Interface` specification; section 42.1.

### 33.1 Subroutines and functions

Fortran has two types of procedures:

- Subroutines, which are somewhat like `void` functions in C++: they can be used to structure the code, and they can only return information to the calling environment through their parameters.
- Functions, which are like C++ functions with a return value.

Both types have the same structure, which is roughly the same as of the main program. For subroutines:

```
subroutine foo( <parameters> )
<variable declarations>
<executable statements>
end subroutine foo
```

and for functions:

```
returntype function foo( <parameters> )
<variable declarations>
<executable statements>
end function foo
```

There is another syntax for declaring functions, see section [33.2.1](#).

Exit from a procedure can happen two ways:

1. the flow of control reaches the end of the procedure body:

```
subroutine foo()
    statement1
    ..
    statementn
end subroutine foo
```

or

2. execution is finished by an explicit `return` statement.

```
subroutine foo()
    print *, "foo"
    if (something) return
    print *, "bar"
end subroutine foo
```

The `return` statement is optional in the first case. The `return` statement is different from C++ in that it does not indicate a returned result value of a function.

**Exercise 33.1.** Rewrite the above subroutine `foo` without a `return` statement.

A subroutine is invoked with a `call` statement:

```
call foo()
```

```
Code:
// funcf/printone.F90
program printone
    implicit none
    call printint(5)
contains
    subroutine printint(invalue)
        implicit none
        integer :: invalue
        print *,invalue
    end subroutine printint
end program printone
```

**Output:**  
[funcf] printone:

5

Arguments types are defined in the body, not the header

```
Code:
// funcf/addone.F90
program addone
    implicit none
    integer :: i=5
    call addint(i,4)
    print *,i
contains
    subroutine addint(inoutvar,addendum)
        implicit none
        integer :: inoutvar,addendum
        inoutvar = inoutvar + addendum
    end subroutine addint
end program addone
```

**Output:**  
[funcf] addone:

9

Parameters are always ‘by reference’!

Recursive functions in Fortran need to be explicitly declared as such, with the `recursive` keyword.

Declare function as `Recursive Function`

**Code:**

```
// funcf/fact.F90
recursive integer function fact(invalue) &
    result (val)
    implicit none
    integer,intent(in) :: invalue
    if (invalue==0) then
        val = 1
    else
        val = invalue * fact(invalue-1)
    end if
end function fact
```

**Output:**

```
[funcf] fact:
echo 7 | ./fact
    7 factorial is
    ↳ 5040
```

Note the `result` clause. This prevents ambiguity.

### 33.2 Return results

While a `subroutine` can only return information through its parameters, a *function* procedure returns an explicit result:

```
logical function test(x)
    implicit none
    real :: x

    test = some_test_on(x)
    return ! optional, see above
end function test
```

You see that the result is not returned in the `return` statement, but rather through assignment to the function name. The `return` statement, as before, is optional and only indicates where the flow of control ends.

A *function* in Fortran is a procedure that return a result to its calling program, much like a non-void function in C++

- `subroutine` vs `function`:  
compare `void` functions vs non-void in C++.
- Function header:  
Return type, keyword `function`, name, parameters
- Function body has statements
- Result is returned by assigning to the function name
- Use: `y = f(x)`

```
Code:
// funcf/plusone.F90
program plussing
    implicit none
    integer :: i
    i = plusone(5)
    print *, i
contains
    integer function plusone(invalue)
        implicit none
        integer :: invalue
        plusone = invalue+1 ! note!
    end function plusone
end program plussing
```

**Output:**  
[funcf] plusone:

6

- The function name is a variable
- ... that you assign to.

A function is not invoked with `call`, but rather through being used in an expression:

```
if (test(3.0) .and. something_else) ...
```

You now have the following cases to make the function known in the main program:

- If the function is in a `contains` section, its type is known in the main program.
- If the function is in a module (see section 37.2 below), it becomes known through a `use` statement.

*F77 note:* Without modules and `contains` sections, you need to declare the function type explicitly in the calling program. The safe way is through using an `interface` specification.

**Exercise 33.2.** Write a program that asks the user for a positive number; non-positive input should be rejected. Fill in the missing lines in this code fragment:

```
Code:
// funcf/readpos.F90
program readpos
    implicit none
    real(4) :: userinput
    print *, "Type a positive number:"
    userinput = read_positive()
    print *, "Thank you for", userinput
contains
    real(4) function read_positive()
        implicit none
        /* ... */
    end function read_positive
end program readpos
```

**Output:**  
[funcf] readpos:

Type a positive number:  
No, not -5.00000000  
No, not 0.00000000  
No, not -3.14000010  
Thank you for 2.48000002

### 33.2.1 The ‘result’ keyword

Apart from assigning to the function name, there is a second mechanism for returning a function result, namely through the `Result` keyword.

```
function some_function() result(x)
  implicit none
  real :: x
  !! stuff
  x = ! some computation
end function
```

You see that here

- the assignment to the name is missing,
- the function name is not typed; but
- instead there is a typed local variable that is marked to be the result.

### 33.2.2 The ‘contains’ clause

<pre>// funcf/nocontain.F90 Program NoContains   implicit none   call DoWhat() end Program NoContains  subroutine DoWhat(i)   implicit none   integer :: i   i = 5 end subroutine DoWhat</pre>	<pre>// funcf/wrongcontain.F90 Program ContainsScope   implicit none   call DoWhat() contains   subroutine DoWhat(i)     implicit none     integer :: i     i = 5   end subroutine DoWhat end Program ContainsScope</pre>
--	---

Warning only, crashes.

Error, does not compile

```
Code:
// funcf/nocontain2.F90
Program NoContainTwo
    implicit none
    integer :: i=5
    call DoWhat(i)
end Program NoContainTwo

subroutine DoWhat(x)
    implicit none
    real :: x
    print *,x
end subroutine DoWhat
```

```
Output:
[funcf] nocontain2.F90:15:16:

    15 |     call DoWhat(i)
        |                 1
Warning: Type mismatch in
    ↪argument 'x' at (1);
    ↪passed INTEGER(4) to
    ↪REAL(4)
    ↪[-Wargument-mismatch]
7.00649232E-45
```

At best compiler warning if all in the same file

### 33.3 Arguments

Arguments are declared in procedure body:

```
subroutine f(x,y,i)
    implicit none
    integer,intent(in) :: i
    real(4),intent(out) :: x
    real(8),intent(inout) :: y
    x = 5; y = y+6
end subroutine f
! and in the main program
call f(x,y,5)
```

declaring the ‘intent’ is optional, but highly advisable.

- Everything is passed by reference.  
Don’t worry about large objects being copied.
- Optional intent declarations:  
Use `in`, `out`, `inout` qualifiers to clarify semantics to compiler.

The term *dummy argument* is what Fortran calls the parameters in the procedure definition:

```
subroutine f(x) ! 'x' is dummy argument
```

The arguments in the procedure call are the *actual arguments*:

```
call f(x) ! 'x' is actual argument
```

Compiler checks your intent against your implementation. This code is not legal:

```
// funcf/intent.F90
subroutine ArgIn(x)
    implicit none
    real,intent(in) :: x
    x = 5 ! compiler complains
end subroutine ArgIn
```

Self-protection: if you state the intended behavior of a routine, the compiler can detect programming mistakes.

Allow compiler optimizations:

```
x = f()
call ArgOut(x)
print *,x
```

Call to f removed

```
do i=1,1000
    x = ! something
    y1 = .... x ....
    call ArgIn(x)
    y2 = ! same expression as y1
```

y2 is same as y1 because x not changed

(May need further specifications, so this is not the prime justification.)

**Exercise 33.3.** Write a subroutine `trig` that takes a number  $\alpha$  as input and passes  $\sin \alpha$  and  $\cos \alpha$  back to the calling environment.

### 33.3.1 Keyword and optional arguments

The arguments in a procedure call can always be given with their corresponding parameter name. This is called a *keyword argument*, and it is sometimes useful to prevent confusion.

```
! confusing:
call two_point( 1.1, 2.2, 3.3, 4.4 )
! better:
call two_point( x1=1.1, x2=2.2, y1=3.3, y2=4.4 )
```

Arguments not given with a keyword are called *positional arguments*. You can mix positional and keyword arguments, but if you give one argument by keyword, all subsequent ones also need their keyword.

- Use the name of the *formal parameter* as keyword.
- Keyword arguments have to come last.

<b>Code:</b> <pre>// funcf/keyword.F90 call say_xy(1,2) call say_xy(x=1,y=2) call say_xy(y=2,x=1) call say_xy(1,y=2) ! call say_xy(y=2,1) ! ILLEGAL contains subroutine say_xy(x,y) implicit none integer,intent(in) :: x,y print *, "x=",x," , y=",y end subroutine say_xy</pre>	<b>Output:</b> <b>[funcf] keyword:</b> <table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td>x=</td><td>1 , y=</td><td>2</td></tr> <tr> <td>x=</td><td>1 , y=</td><td>2</td></tr> <tr> <td>x=</td><td>1 , y=</td><td>2</td></tr> <tr> <td>x=</td><td>1 , y=</td><td>2</td></tr> </table>	x=	1 , y=	2									
x=	1 , y=	2											
x=	1 , y=	2											
x=	1 , y=	2											
x=	1 , y=	2											

A relation notion is that of *optional arguments*. A parameter can be marked **optional**, after which it can be omitted from a procedure call.

- Optional parameters can be anywhere in the parameter list;
- If you omit one optional parameter in the argument list, all subsequent arguments need to be given by keyword.
- The procedure can test whether or not an optional argument was supplied with the function **Present**

- Extra specifier: **Optional**
  - Presence of argument can be tested with **Present**

## 33.4 Types of procedures

Procedures that are in the main program (or another type of program unit), separated by a **contains** clause, are known as *internal procedures*. This is as opposed to *module procedures*.

There are also *statement functions*, which are single-statement functions, usually to identify commonly used complicated expressions in a program unit. Presumably the compiler will *inline* them for efficiency.

The standard library functions, such as **sqrt**, can be declared as such in an **intrinsic** statement

```
Intrinsic :: sqrt,cmplx
```

but this is not necessary.

The **entry** statement is so bizarre that I refuse to discuss it.

## 33.5 Local variable **save-ing**

Normally, local variables in a procedure act as if they get created when the procedure is invoked, and disappear again when its execution ends. It is possible to retain the value of a variable between invocations by giving it an attribute of **save**.

```
subroutine whatever()
integer,save :: i
```

### 33. Procedures

---

(This corresponds roughly to a *static* variable in C++.)

Here is a major pitfall. If you give a local variable an initialization value:

```
subroutine whatever()
integer :: i = 5
```

then the variable implicitly gets a *save* attribute, whether this is specified or not. The initialization is only executed once, probably at compile time, and at the second procedure invocation the saved value is used.

This may trip you up as the following example shows:

Local variable is initialized only once,  
second time it uses its retained value.

**Code:**

```
// funcf/save.F90
integer function maxof2(i, j)
  implicit none
  integer, intent(in) :: i, j
  integer :: max=0
  if (i>max) max = i
  if (j>max) max = j
  maxof2 = max
end function maxof2
```

**Output:**

```
[funcf] save:
Comparing: 1   3
           3
Comparing: -2 -4
           3
```

# Chapter 34

## Scope

### 34.1 Scope

Fortran ‘has no curly brackets’: you not easily create nested scopes with local variables as in C++. For instance, the range between `do` and `end do` is not a scope. This means that all variables have to be declared at the top of a program or subprogram.

#### 34.1.1 Variables local to a program unit

Variables declared in a subprogram have similar scope rules as in C++:

- Their visibility is controlled by their textual scope:

```
Subroutine Foo()
    integer :: i
    ! 'i' can now be used
    call Bar()
    ! 'i' still exists
End Subroutine Foo
Subroutine Bar() ! no parameters
    ! The 'i' of Foo is unknown here
End Subroutine Bar
```

- Their dynamic scope is the lifetime of the program unit in which they are declared:

```
Subroutine Foo()
    call Bar()
    call Bar()
End Subroutine Foo
Subroutine Bar()
    Integer :: i
    ! 'i' is created every time Bar is called
End Subroutine Bar
```

(That last example has a little subtlety; see section 33.5 for the `Save` attribute on procedure variables.)

##### 34.1.1.1 Variables in a module

Variables in a module (section 37.2) have a lifetime that is independent of the calling hierarchy of program units: they are *static variables*.

#### 34.1.1.2 Other mechanisms for making static variables

Before Fortran gained the facility for recursive functions, the data of each function was placed in a statically determined location. This meant that the second time you call a function, all variables still have the value that they had last time. To force this behavior in modern Fortran, you can add the *Save* specification to a variable declaration.

Another mechanism for creating static data was the `Common` block. This should not be used, since a `Module` is a more elegant solution to the same problem.

#### 34.1.2 Variables in an internal procedure

An *internal procedure* (that is, one placed in the `Contains` part of a program unit) can receive arguments from the containing program unit. It can also access directly any variable declared in the containing program unit, through a process called *host association*.

The rules for this are messy, especially when considering implicit declaration of variables, so we advise against relying on it.

# Chapter 35

## String handling

### 35.1 String denotations

A string can be enclosed in single or double quotes. That makes it easier to have the other type in the string.

```
// stringf/quote.F90
print *, 'This string was in single quotes'
print *, 'This string in single quotes contains a single '' quote'
print *, "This string was in double quotes"
print *, "This string in double quotes contains a double "" quote"
```

### 35.2 Characters

The datatype `Character` is used both for characters and strings. Therefore, see next section.

### 35.3 Strings

The length of a Fortran string is specified with the `len` keyword when the string is created:

```
character(len=50) :: mystring
mystring = "short string"
```

The `len` function also gives the length of the string, but note that that is the length with which it was allocated, not how much non-blank content you put in it.

String length, with / without trimming.

## 35. String handling

```
Code:  
// stringf/quote.F90  
character(len=12) :: strvar  
/* ... */  
strvar = "word"  
print *, len(strvar), len(trim(strvar))
```

```
Output:  
[stringf] strlen:  
12 4
```

To get the more intuitive length of a string, that is, the location of the last non-blank character, you need to **trim** the string.

Concatenation is done with a double slash:

```
Code:  
// stringf/quote.F90  
character(len=10) :: firstname, lastname  
character(len=15) :: shortname, fullname  
/* ... */  
firstname = "Victor"; lastname = "Eijkhout"  
shortname = firstname // lastname  
print *, "without trimming: ", shortname  
fullname = trim(firstname) // trim(lastname)  
print *, "with trimming: ", fullname
```

```
Output:  
[stringf] concat:  
without trimming: Victor  
→ Eijkh  
with trimming: Victor Eijkhout
```

## 35.4 Conversions

Sometimes we want to convert between the string 123 and the number 123. Let's start easy, by looking at characters and their *ascii* codes.

### 35.4.1 Character conversions

Given an integer, **Char** gives the character with that ascii code. That can be a printable or an unprintable character:

```
Code:  
// stringf/convert.F90  
print *, "97 is a:", char(97)  
print *, "84 is T:", char(84)  
print *, "53 is 5:", char(53)  
print *, "11 is VT :", char(11), ". "
```

```
Output:  
[stringf] ascii:  
97 is a:a  
84 is T:T  
53 is 5:5  
11 is VT :  
.
```

Note the last one!

In the other direction, **Iachar** gives the ascii code of a character.

```

character :: char
integer :: code

char = "x"
code = iachar(char)
print *,char," has code",code

```

**Remark** There is also a function *Ichar*, but it returns the code in the native character set. In rare cases this can be something other than ascii.

**Exercise 35.1.** Write a test to see if a character is lowercase:

**Code:**

```

// stringf/convert.F90
print *, "lower t", islower("t")
print *, "lower T", islower("T")
print *, "lower 3", islower("3")

```

**Output:**

```

[stringf] lower:
lower t T
lower T F
lower 3 F

```

Similarly, write a test *isdigit*.

### 35.4.2 String conversions

Converting between a number and a string relies on concept from the I/O chapter (chapter 41); see section 41.5.

## 35.5 Further notes

In addition to the character definition with the *len* specification, there is

```

character*80 :: str
character,dimension(80) :: str
character :: str(80)

```

These should not be used.



# Chapter 36

## Structures, eh, types

Fortran has structures for bundling up data, but there is no `struct` keyword: instead you declare both the structure type and variables of that *derived type* with the `type` keyword.

### 36.1 Derived type basics

Now you need to

- Define the type to describe what's in it;
- Declare variables of that type; and
- use those variables, but setting the type members or using their values.

`Type name / End Type name` block.

Member declarations inside the block:

```
type mytype
    integer :: number
    character :: name
    real(4) :: value
end type mytype
```

Type definitions go before executable statements.

Creating type variables is a little different from objects in a C++ class. In C++ the class name could be used by itself as the datatype; in Fortran you need to write `Type (mytype)`. Otherwise, it looks like any other variable declaration.

Declare type variables in the main program:

```
Type (mytype) :: struct1, struct2
```

Initialize with type name:

```
struct1 = mytype( 1, 'my_name', 3.7 )
```

Copying:

```
struct2 = struct1
```

If you need access to a single field in a type, there is a notation analogous to the ‘dot’ notation in C++: in Fortran you use the percent sign %.

Access structure members with %  
(compare C++ dot-notation)

```
Type(mytype) :: typed_struct
typed_struct%member = ....
```

As an example, we use the ‘point’ structure from the geometry project.

<pre>// structf/pointtype.F90 type point     real :: x,y end type point</pre>	<pre>// structf/pointtype.F90 type(point) :: p1,p2 p1 = point(2.5, 3.7)  p2 = p1 print *,p1 print *,p2%x,p2%y</pre>
---	---

Note that printing a type by itself is equivalent to printing its components in sequence.

You can have arrays of types:

```
type(my_struct) :: data
type(my_struct),dimension(10) :: data_array
```

## 36.2 Derived types and procedures

Structures can be passed as procedure argument, just like any other datatype. In this example the function *length*:

- Takes a structure of `type(point)` as argument; and
- returns a `real(4)` result.
- The structure is declared as `intent(in)`.

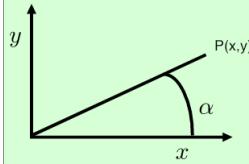
Function with structure argument:

```
// structf/pointtype.F90
real(4) function length(p)
    implicit none
    type(point),intent(in) :: p
    length = sqrt( &
                    p%x**2 + p%y**2 )
end function length
```

Function call

```
// structf/pointtype.F90
print *, "Length:", length(p2)
```

**Exercise 36.1.** Add a function `angle` that takes a `Point` argument and returns the angle of the  $x$ -axis and the line from the origin to that point.



Your program should read in the  $x, y$  values of the point and print out the angle in radians.

Bonus: can you print the angle as a fraction of  $\pi$ ? So

$$(1, 1) \Rightarrow 0.25$$

You can base this off the file `point.F90` in the repository

**Exercise 36.2.** Write a program that has the following:

- A type `Point` that contains real numbers  $x, y$ ;
- a type `Rectangle` that contains two `Points`, corresponding to the lower left and upper right point;
- a function `area` that has one argument: a `Rectangle`.

Your program should

- Accept two real numbers on one line, for the bottom left point;
- similarly, again on one line, the coordinates of the top right point; then
- print out the area of the (axi-parallel) rectangle defined by these two points.

**Exercise 36.3.** In the previous exercise 36.2:

Bonus points for using a module,

double bonus points for using an object-oriented solution.

### 36.3 Parameterized types

If a derived type contains an array, you may want to have the length of that array to be variable, without making the array dynamically allocatable. For this, Fortran has *parameterized types*: you can define a type with some combination of:

- a parameter with attribute `len`, used as the length of an array member; or
- a parameter with attribute `kind`, used as the kind of some variable; section 30.4.1.2.

Example:

```
// structf/parampoint.F90
type point(dim)
    integer,len :: dim
    real,dimension(dim) :: x
end type point
```

I haven't figured out how to set variables:

```
// structf/parampoint.F90
type(point(3)) :: p1,p2
p1%x = [1.,2.,3.]
p2 = p1
print *,p2%x
```

These types can be passed normally:

```
// structf/parampoint.F90
real(4) function length(p)
    implicit none
    type(point(3)),intent(in) :: p
    length = sqrt( &
        p%x(1)**2 + p%x(2)**2 + p%x(3)**2 )
end function length
```

## Chapter 37

### Modules

Fortran has a clean mechanism for importing data (including numeric constants such as  $\pi$ ), functions, types that are defined in another file. This is done through modules, defined with the `module` keyword.

Modules look like a program, but without main  
(only ‘stuff to be used elsewhere’):

```
// structf/typemod.F90
Module geometry
    type point
        real :: x,y
    end type point
    real(8),parameter :: pi = 3.14159265359
contains
    real(4) function length(p)
        implicit none
        type(point),intent(in) :: p
        length = sqrt( p%x**2 + p%y**2 )
    end function length
end Module geometry
```

Note also the numeric constant.

Module imported through `use` statement;  
placed before `implicit none`

<b>Code:</b> <pre>// structf/typemod.F90 Program size use geometry implicit none  type(point) :: p1,p2 p1 = point(2.5, 3.7)  p2 = p1 print *,p2%x,p2%y print *, "length:", length(p2) print *, 2*pi  end Program size</pre>	<b>Output:</b> <b>[structf] typemod:</b> <pre>2.50000000      3.70000005 length:   4.46542263 6.2831854820251465</pre>
--	--

**Exercise 37.1.** Take exercise 36.2 and put all type definitions and all functions in a module.

## 37.1 Modules for program modularization

Modules are Fortran's mechanism for supporting *separate compilation*: you can put your module in one file, your main program in another, and compile them separately.

A module is a container for definitions of subprograms and types, and for data such as constants and variables. A module is not a structure or object: there is only one instance.

**Remark** The `use` statement is somewhat similar to an `#include "stuff.h"` line in C++. However, note that C++20 has also adopted modules, as cleaner than preprocessor-based solutions.

## 37.2 Module definition

What do you use a module for?

- Type definitions: it is legal to have the same type definition in multiple program units, but this is not a good idea. Write the definition just once in a module and make it available that way.
- Function definitions: this makes the functions available in multiple sources files of the same program, or in multiple programs.
- Define constants: for physics simulations, put all constants in one module and use that, rather than spelling out the constants each time.
- Global variables: put variables in a module if they do not fit an obvious scope.

Any routines come after the `contains` clause.

**Remark** Modules were introduced in Fortran90. In earlier standards, information could be made globally available through `common` blocks. Since modules are much cleaner than common blocks, do not use those anymore.

A module is made available with the `use` keyword, which needs to go before the `implicit none`.

```
Use statement placed before Implicit
// structf/module.F90
Program ModProgram
    use FunctionsAndValues
    implicit none

    print *, "Pi is:", pi
    call SayHi()

End Program ModProgram
```

Also possible:

```
Use mymodule, Only: func1, func2
Use mymodule, func1 => new_name1
```

If you compile a module, you will find a .mod file in your directory. (This is little like a .h file in C++.) If this file is not present, you can not **use** the module in another program unit, so you need to compile the file containing the module first.

**Exercise 37.2.** Write a module *PointMod* that defines a type *Point* and a function *distance* to make this code work:

```
// geomf/pointmain.F90
use pointmod
implicit none
type(Point) :: p1, p2
real(8) :: p1x, p1y, p2x, p2y
read *, p1x, p1y, p2x, p2y
p1 = point(p1x, p1y)
p2 = point(p2x, p2y)
print *, "Distance:", distance(p1, p2)
```

Put the program and module in two separate files and compile thusly:

```
ifort -g -c pointmod.F90
ifort -g -c pointmain.F90
ifort -g -o pointmain pointmod.o pointmain.o
```

### 37.3 Separate compilation

The exercises in this course are simple enough that you can include any modules in the same file as your main program. However, in realistic applications you will have a separate files for modules, maybe even using one file per module.

Suppose program is split over two files:

`theprogram.F90` and `themodule.F90`.

- Compile the module: `ifort -c themodule.F90`; this gives
  - an *object file* (extension: `.o`) that will be linked later, and
  - a module file `modulename.mod`.
- Compile the main program:  
`ifort -c theprogram.F90` will read the `.mod` file; and finally
- Link the object files into an *executable*:  
`ifort -o myprogram theprogram.o themodule.o`  
 The compiler is used as *linker*: there is no compiling in this step.

Important: the module needs to be compiled before any (sub)program that uses it.

The Fortran2008 standard introduced *sub modules*, which can even further facilitate separate compilation.

## 37.4 Access

By default, all the contents of a module is usable by a subprogram that uses it. However, a keyword `private` make module contents available only inside the module. You can make the default behavior explicit by using the `public` keyword. Both `public` and `private` can be used as attributes on definitions in the module. There is a keyword `protected` for data members that are public, but can not be altered by code outside the module.

```
// module/protect.F90
Module settings
    implicit none                                // module/protect.F90
    logical,protected :: has_initialized           ! has_initialized ) then
contains                                         call init()
    subroutine init()                            end if
        has_initialized = .TRUE.    !! WRONG does not compile:
    end subroutine init                         ! has_initialized = .FALSE.
End Module settings
```

## 37.5 Polymorphism

```
module somemodule

INTERFACE swap
MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
END INTERFACE

contains
subroutine swapreal
...
end subroutine swapreal
subroutine swapint
...
end subroutine swapint
```

## 37.6 Operator overloading

You can define operations such as + or \* on types.

```
// typef/plustype.F90
Module TypeDef
    Type inttype
        integer :: value
    end type inttype
    Interface operator(+)
        module procedure addtypes
    end Interface operator(+)
contains
```

```
function addtypes(i1,i2) result(isum)
    implicit none
    Type(inttype),intent(in) :: i1,i2
    Type(inttype) :: isum
    isum%value = i1%value+i2%value
end function addtypes
end Module TypeDef
```

You can now make the code look nice and simple:

<b>Code:</b>	<b>Output:</b>
<pre>// typef/plustype.F90 Type(inttype) :: i1,i2,i3 i1 = inttype(1); i2 = inttype(2) i3 = i1+i2 print *, "Sum:", i3%value</pre>	<b>[typef] plus:</b> <i>Sum:</i> 3

Overloading includes the assignment operator:

```
INTERFACE ASSIGNMENT (=)
subroutine_interface_body
END INTERFACE
```

You can define new operators with a dot-notation:

```
INTERFACE OPERATOR (.DIST.)
MODULE PROCEDURE calcdist
END INTERFACE
```



# Chapter 38

## Classes and objects

### 38.1 Classes

Fortran classes are based on `type` objects. Some aspects are similar to C++. For instance, the same syntax is used for specifying data members and methods:

```
print *,myobject%xfield  
myobject%set_xfield(5.1)
```

Other aspects are a little different: in C++ you can write in one class definition all data and function members; in Fortran data and functions are declared separately.

A big difference is in how function methods are defined: the object itself becomes an extra parameter. You will see the details later.

First about how Fortran classes are organized. A class is a type definition inside a module, with an extra clause indicating what function methods are available for the type.

You define a type as before, with its data members, but now the type has a `contains` for the methods:

```
// objectf/mult1.F90  
Module multmod  
  
    type Scalar  
        real(4) :: value  
        contains  
            procedure,public :: &  
                printme,scaled  
    end type Scalar  
  
    contains ! methods  
        /* ... */  
    end Module multmod
```

As stated above, calling methods on an object uses the same syntax as accessing its data members.

Method call similar to C++

## 38. Classes and objects

---

**Code:**

```
// objectf/mult1.F90
Program Multiply
use multmod
implicit none

type(Scalar) :: x
real(4) :: y
x = Scalar(-3.14)
call x%printme()
y = x%scaled(2.)
print '(f7.3)',y

end Program Multiply
```

**Output:**

```
[objectf] mult1:
The value is -3.140
-6.280
```

The method definition works slightly different from C++, but if you know python you'll see the similarity. If a method is called with one argument;

```
call obj%fun(arg)
```

the function has two parameters, the first one being the object, and the second one the parenthesized argument.

Additionally, the first parameter is of type **Type** (*obj*), but in the method it is declared as **Class** (*obj*).

Note the extra first parameter:

which is a **Type** but declared here as **Class**:

```
// objectf/mult1.F90
subroutine printme(me)
    implicit none
    class(Scalar) :: me
    print '("The value is",f7.3)',me%value
end subroutine printme
function scaled(me,factor)
    implicit none
    class(Scalar) :: me
    real(4) :: scaled,factor
    scaled = me%value * factor
end function scaled
```

In summary:

- A class is a **Type** with a **contains** clause followed by **procedure** declarations,
- ... contained in a module.
- Actual methods go in the **contains** part of the module
- First argument of method is the object itself.

```
// geomf/pointexample.F90
Module PointClass
    Type,public :: Point
        real(8) :: x,y
    contains
        procedure, public :: &
            distance
    End type Point
    contains
        !! ... distance function ...
        /* ... */
End Module PointClass
// geomf/pointexample.F90
Program PointTest
    use PointClass
    implicit none
    type(Point) :: p1,p2
    p1 = point(1.d0,1.d0)
    p2 = point(4.d0,5.d0)
    print *, "Distance:",&
        p1%distance(p2)
End Program PointTest
```

	C++	Fortran
Members	in the object	in the ‘type’
Methods	in the object	interface: in the type implementation: in the module
Constructor	default or explicit	none
object itself	‘this’	first argument
Class members	global variable	accessed through first arg
Object’s methods	period	percent

**Exercise 38.1.** Take the point example program and add a distance function:

```
Type(Point) :: p1,p2
! ... initialize p1,p2
dist = p1%distance(p2)
! ... print distance
```

You can base this off the file *pointexample.F90* in the repository

**Exercise 38.2.** Write a method add for the Point type:

```
Type(Point) :: p1,p2,sum
! ... initialize p1,p2
sum = p1%add(p2)
```

What is the return type of the function add?

### 38.1.1 Final procedures: destructors

The Fortran equivalent of *destructors* is a *final procedure*, designated by the *final* keyword.

```
// objectf/final.F90
contains
    final :: &
        print_final
end type Scalar
```

A final procedure has a single argument of the type that it applies to:

```
// objectf/final.F90
subroutine print_final(me)
    implicit none
    type(Scalar) :: me
    print '("On exit: value is",f7.3)',me%value
end subroutine print_final
```

The final procedure is invoked when a derived type object is deleted, except at the conclusion of a program:

```
// objectf/final.F90
call tmp_scalar()
contains
    subroutine tmp_scalar()
        type(Scalar) :: x
        real(4) :: y
        x = Scalar(-3.14)
    end subroutine tmp_scalar
```

## 38.2 Inheritance

Inheritance:

```
type,extends(baseclas) :: derived_class
```

Pure virtual:

```
type,abstract
```

<http://fortranwiki.org/fortran/show/Object-oriented+programming>

It is of course best to put the type definition and method definitions in a module, so that you can `use` it.

Mark methods as `private` so that they can only be used as part of the `type`:

```
// geomf/point.F90
Module PointClass
    /* ... */
    private
contains
    subroutine setzero(p)
        implicit none
        class(point) :: p
        p%x = 0.d0 ; p%y = 0.d0
    end subroutine setzero
```

```
/* ... */
End Module PointClass
```

### 38.3 Operator overloading

For many physical quantities it makes sense to define an addition operator. This makes it possible to write

```
Type(X) :: x,y,z
! stuff
x = y+z
```

For purposes of exposition, let's make a very simple class:

```
// geomf/scalar.F90
Type,public :: ScalarField
    real(8) :: value
contains
    procedure,public :: set,print
    procedure,public :: add
End type ScalarField
```

We define a couple of obvious methods:

```
// geomf/scalar.F90
subroutine set(v,x)
    implicit none
    class(ScalarField) :: v
    real(8),intent(in) :: x
    v%value = x
end subroutine set

subroutine print(v)
    implicit none
    class(ScalarField) :: v
    print '(f7.4)', v%value
end subroutine print

// geomf/scalar.F90
call u%set(2.d0)
call v%set(1.d0)
! z = u%add(v)
z = u+v
```

Before we can define the addition operator, it is first necessary to define an addition function:

```
// geomf/scalar.F90
function add(in1,in2) result(out)
    implicit none
    class(ScalarField),intent(in) :: in1
    type(ScalarField),intent(in) :: in2
    type(ScalarField) :: out
```

```

        out%value = in1%value + in2%value
    end function add

```

This function needs to satisfy some conditions:

- The function needs to have two input parameters. Obviously.
- The input parameters need to be declared **Intent (In)**. This is a little less obvious, but it makes sense, because the arguments to the addition parameter are not really passed the normal way.

Turning the function into an operator is then pretty simple.

Interface block:

```

// geomf/scalar.F90
interface operator(+)
    module procedure add
end interface operator(+)

```

**Exercise 38.3.** Extend the above example program so that the type stores an array instead of a scalar.

**Code:**

```

// geomf/field.F90
integer,parameter :: size = 12

Type(VectorField) :: u,v,z

call u%alloc(size)
call v%alloc(size)
call u%setlinear()
call v%setconstant(1.d0)
!   z = u%add(v)
z = u+v
call z%print()

```

**Output:**

[geomf] field:
2.0000 3.0000 4.0000 5.0000
↔ 6.0000 7.0000 8.0000
↔ 9.0000 10.0000 11.0000
↔ 12.0000 13.0000

You can base this off the file *scalar.F90* in the repository

Similarly, we can redefine the assignment operator; see <https://dannyvanpoucke.be/oop-fortran-tut5-en/>. This comes with some complications regarding *shallow copy* and *deep copy*.

## Chapter 39

### Arrays

Array handling in Fortran is similar to C++ in some ways, but there are differences, such as that Fortran indexing starts at 1, rather than 0. More importantly, Fortran has better handling of multi-dimensional arrays, and it is easier to manipulate whole arrays.

#### 39.1 Static arrays

The preferred way for specifying an array size is:

Creating arrays through `dimension` keyword:

```
real(8), dimension(100) :: x,y
```

One-dimensional arrays of size 100.

```
integer,dimension(10,20) :: iarr
```

Two-dimensional array of size  $10 \times 20$ .

These arrays are statically defined, and only live inside their program unit (subroutine, function, module).

Dynamic allocation later.

Such an array, with size explicitly indicated, is called a *static array* or *automatic array*. (See section 39.4 for dynamic arrays.)

Array indexing in Fortran is 1-based by default:

```
integer,parameter :: N=8
real(4),dimension(N) :: x
do i=1,N
  ... x(i) ...
end do
```

Different from C/C++.

Note the use of `parameter`: *compile-time constant*  
Size needs to be known to the compiler.

Unlike C++, Fortran can specify the lower bound explicitly:

```
real, dimension(-1:7) :: x
do i=-1,7
  ... x(i) ...
```

Preferred: use **lbound** and **ubound**

(see also 39.2.1)

**Code:**

```
// arrayf/query.F90
real, dimension(-1:7) :: array
integer :: idx
/* ... */
do idx=lbound(array,1),ubound(array,1)
  array(idx) = 1+idx/10.
  print *,array(idx)
end do
```

**Output:**

[arrayf]	<b>lubound:</b>
0.899999976	
1.00000000	
1.10000002	
1.20000005	
1.29999995	
1.39999998	
1.50000000	
1.60000002	
1.70000005	

Such arrays, as in C++, obey the scope: they disappear at the end of the program or subprogram.

### 39.1.1 Initialization

There are various syntaxes for *array initialization*, including the use of *implicit do-loops*:

Different syntaxes:

- Explicit:

```
// arrayf/init.F90
real, dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]
```

- Implicit do-loop:

```
// arrayf/init.F90
real5 = [ (1.01*i,i=1,size(real5,1)) ]
```

- Legacy syntax

```
// arrayf/init.F90
real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
```

(This is pre-Fortran2003. Slashes were also used for some other deprecated constructs.)

### 39.1.2 Array sections

Fortran is more sophisticated than C++ in how it can handle arrays as a whole. For starters, you can assign one array to another:

```
real*8, dimension(10) :: x,y
x = y
```

This obviously requires the arrays to have the same size. You can assign subarrays, or *array sections*, as long as they have the same shape. This uses a colon syntax.

- $A(:)$  to get all indices,
- $A(:n)$  to get indices up to  $n$ ,
- $A(n:)$  to get indices  $n$  and up.
- $A(m:n)$  indices in range  $m, \dots, n$ .

Assignment from one section to another:

**Code:**

```
// arrayf/section.F90
real(8), dimension(5) :: x = &
    [.1d0, .2d0, .3d0, .4d0, .5d0]
    /* ... */
x(2:5) = x(1:4)
print '(f5.3)', x
```

**Output:**

```
[arrayf] sectionassign:
0.100
0.100
0.200
0.300
0.400
```

Note:

Format syntax will be discussed later:  
float number, 5 positions, 3 after decimal point.

**Exercise 39.1.** Code out the array assignment

```
x(2:5) = x(1:4)
```

with an explicit indexed loop. Do you get the same output? Why? What conclusion do you draw about internal mechanisms used in array sections?

The above exercise illustrates a point about the *semantics of array operations*: an array statement behaves as if all inputs are gathered together before any results are stored. Conceptually, it is as if the right-hand side is assembled and copied to some temporary locations before being written to the left-hand side. In practice, this may require large temporary arrays (and negatively affect performance by lessening *locality*) so you hope that the compiler does something smarter. However, the exercise showed that an array assignment can not trivially be converted to a simple loop.

**Exercise 39.2.** Can you formalize the sort of array statement for which a simple translation to a loop changes the semantics? (In compiler terminology this is called a *dependence*.)

Array operations can be more sophisticated than assigning to a whole array or a section of it. For instance, you can use a stride:

```
X(a:b:c) : stride c
```

Analogous to: `do i=a, b, c`

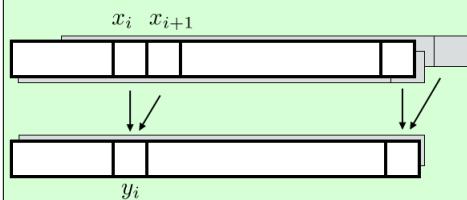
Copy a contiguous array to a strided subset of another:

```
Code:
// arrayf/section.F90
integer, dimension(5) :: &
    y = [0,0,0,0,0]
integer, dimension(3) :: &
    z = [3,3,3]
/* ... */
y(1:5:2) = z(:)
print '(i3)', y
```

```
Output:
[arrayf] sectionmg:
3
0
3
0
3
```

You can even do arithmetic on array sections, for instance adding them together.

**Exercise 39.3.** Code  $\forall_i: y_i = (x_i + x_{i+1})/2$ :



- First with a do loop; then
- in a single array assignment statement by using sections.

Initialize the array  $x$  with values that allow you to check the correctness of your code.

### 39.1.3 Integer arrays as indices

It's even possible to use a set of indices, stored in an integer array, to access arbitrary locations in an array.

Indexed subset:

```
integer, dimension(4) :: i = [2,3,5,7]
real(4), dimension(10) :: x
print *, x(i)
```

## 39.2 Multi-dimensional

Arrays above had ‘rank one’. The rank is defined as the number of indices you need to address the elements. Mathematically this is not a rank but the dimension of the array, but that word is already taken. We will still use that word, for instance talking about the first and second dimension of an array.

A rank-two array, or matrix, is defined like this:

Declaration and use with parentheses and comma  
(compare  $a[i][j]$  in C++):

```
real(8),dimension(20,30) :: array
array(i,j) = 5./2
```

A useful function is `reshape`.

Reshape: convert 2D array to 1D (or vv)  
between arrays with the same number of elements.

Example:

- initialize as 1D,
- reshape to 2D

**Code:**

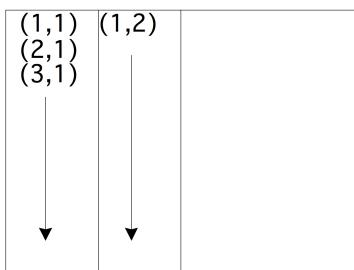
```
// arrayf/multi.F90
real,dimension(2,2) :: x
x = reshape( [ ( 1.*i,i=1,size(x) ) ], shape(x)
    )
print *,x
```

**Output:**

[arrayf] multi:	1.00000000	2.00000000
	↪ 3.00000000	
	↪ 4.00000000	

With multidimensional arrays we have to worry how they are stored in memory. Are they stored row-by-row, or column-by-column? In Fortran the latter choice, also known as *column-major* storage, is used; see figure 39.1.

Fortran column major



Physical:

(1,1)	(2,1)	(3,1)	...	(1,2)	...
-------	-------	-------	-----	-------	-----

Figure 39.1: Column-major storage in Fortran

To traverse the elements as they are stored in memory, you would need the following code:

```
do col=1,size(A,2)
  do row=1,size(A,1)
    .... A(row,col) .....
  end do
end do
```

This is sometimes described as ‘First index varies quickest’. There are various performance-related reasons why such traversal is better than with the loops exchanged.

**Exercise 39.4.** Can you describe in words how memory elements are accessed if you would write

```
do row=1,size(A,1)
  do col=1,size(A,2)
```

## 39. Arrays

---

```
.... A(row,col) ....  
end do  
end do  
  
?
```

You can make sections in multi-dimensional arrays: you need to indicate a range in all dimensions.

```
real(8),dimension(10) :: a,b  
a(1:9) = b(2:10)
```

or

```
logical,dimension(25,3) :: a  
logical,dimension(25)    :: b  
a(:,2) = b
```

You can also use strides.

Fill array by rows, printing is by column:

$$\begin{pmatrix} 1 & 2 & \dots & N \\ N+1 & \dots & & \\ & \dots & & MN \end{pmatrix}$$

**Code:**

```
// arrayf/printarray.F90  
integer,parameter :: M=4,N=5  
real(4),dimension(M,N) :: rect  
  
do i=1,M  
  do j=1,N  
    rect(i,j) = count  
    count = count+1  
  end do  
end do  
print *,rect
```

**Output:**

**[arrayf] printarray:**

```
1.00000000      6.00000000  
  ↳       11.0000000  
  ↳       16.0000000  
  ↳       2.00000000  
  ↳       7.00000000  
  ↳       12.0000000  
  ↳       17.0000000  
  ↳       3.00000000  
  ↳       8.00000000  
  ↳       13.0000000  
  ↳       18.0000000  
  ↳       4.00000000  
  ↳       9.00000000  
  ↳       14.0000000  
  ↳       19.0000000  
  ↳       5.00000000  
  ↳       10.0000000  
  ↳       15.0000000  
  ↳       20.0000000
```

### 39.2.1 Querying an array

We have the following properties of an array:

- The bounds are the lower and upper bound in each dimension. For instance, after

```
integer, dimension(-1:1, -2:2) :: symm
```

the array `symm` has a lower bound of `-1` in the first dimension and `-2` in the second. The functions `Lbound` and `Ubound` give these bounds as array or scalar:

```
array_of_lower = Lbound(symm)
upper_in_dim_2 = Ubound(symm, 2)
```

#### Code:

```
// arrayf/section.F90
real(8), dimension(2:N+1) :: Afrom2 = &
    [1, 2, 3, 4, 5]
/* ... */
lo = lbound(Afrom1, 1)
hi = ubound(Afrom1, 1)
print *, lo, hi
print '(i3,":",f5.3)', &
    (i, Afrom1(i), i=lo, hi)
```

#### Output:

[arrayf] fsection2:

1	5
1:1.000	
2:2.000	
3:3.000	
4:4.000	
5:5.000	

- The `extent` is the number of elements in a certain dimension, and the `shape` is the array of extents.
- The `size` is the number of elements, either for the whole array, or for a specified dimension.

```
integer :: x(8), y(5,4)
size(x)
size(y, 2)
```

### 39.2.2 Reshaping

#### RESHAPE

```
array = RESHAPE( list, shape )
```

Example:

```
// arrayf/shape.F90
square = reshape( ((i, i=1, 16)), (/4, 4/) )
```

#### SPREAD

```
array = SPREAD( old, dim, copies )
```

## 39.3 Arrays to subroutines

Subprogram needs to know the shape of an array, not the actual bounds:

Passing array as one symbol:

**Code:**

```
// arrayf/arraypass1d.F90
real(8), dimension(:) :: x(N) &
    = [ (i, i=1, N) ]
real(8), dimension(:) :: y(0:N-1) &
    = [ (i, i=1, N) ]

sx = arraysuum(x)
sy = arraysuum(y)
print '(\"Sum of one-based array:\",/,4x,f6.3)', 
      sx
print '(\"Sum of zero-based array:\",/,4x,f6.3)
      ', sy
```

**Output:**  
**[arrayf] arraypass1d:**

```
Sum of one-based array:
55.000
Sum of zero-based array:
55.000
```

Note declaration as `dimension(:)`  
actual size is queried

```
// arrayf/arraypass1d.F90
real(8) function arraysuum(x)
    implicit none
    real(8), intent(in), dimension(:) :: x
    real(8) :: tmp
    integer i

    tmp = 0.
    do i=1,size(x)
        tmp = tmp+x(i)
    end do
    arraysuum = tmp
end function arraysuum
```

The array inside the subroutine is known as a *assumed-shape array* or *automatic array*.

## 39.4 Allocatable arrays

Static arrays are fine at small sizes. However, there are two main arguments against using them at large sizes.

- Since the size is explicitly stated, it makes your program inflexible, requiring recompilation to run it with a different problem size.
- Since they are allocated on the so-called *stack*, making them too large can lead to *stack overflow*.

A better strategy is to indicate the shape of the array, and use `allocate` to specify the size later, presumably in terms of run-time program parameters.

```

! static:
integer,parameter :: s=100
real(8), dimension(s) :: xs,ys

! dynamic
integer :: n
real(8), dimension(:), allocatable :: xd,yd
read *,n
allocate(xd(n), yd(n))

```

You can `deallocate` the array when you don't need the space anymore.

If you are in danger of running out of memory, it can be a good idea to add a `stat=ierror` clause to the `allocate` statement:

```

integer :: ierr
allocate( x(n), stat=ierr )
if ( ierr/=0 ) ! report error

```

Has an array been allocated:

`Allocated( x )` ! returns logical

Allocatable arrays are automatically deallocated when they go out of scope. This prevents the *memory leak* problems of C++.

Explicit deallocate:

`deallocate(x)`

### 39.4.1 Returning an allocated array

In an effort to keep the main program nice and abstract, you may want to delegate the `allocate` statement to a procedure. In case of a `Subroutine`, you can pass the (unallocated) array as a parameter. But can you return it from a `Function`?

This requires the `Result` keyword (section 33.2.1):

```

// arrayf/returnarray.F90
function create_array(n) result(v)
    implicit none
    integer,intent(in) :: n
    real,dimension(:),allocatable :: v
    integer :: i
    allocate(v(n))
    print *, "allocated with shape:", shape(v)
    v = [ (i+.5,i=1,n) ]
end function create_array

```

## 39.5 Array output

The simple statement

```
print *,A
```

will output the element of `A` in memory order; see section 39.2.

For a more sophisticated approach, the main thing to know is that each call to a format statement starts on a new line. Thus

```
print '(10f7.3)',A( ...stuff... )
```

will print 10 elements of the array on each line, before starting a new line of output.

The way to specify the elements of the array is to use implicit do-loops; section 32.3:

```
print '(10f7.3)', (A(i), i=1, size(A))
```

or for multiple dimensions:

```
do row=1, size(A, 2)
   print '(10f7.3)', (A(i, row), i=1, size(A, 1))
end do
```

What if, in this example, the rows are longer than 10 elements? You can not parametrize the format, but there is no harm in specifying more format than there are array elements:

## 39.6 Operating on an array

### 39.6.1 Arithmetic operations

Between arrays of the same shape:

```
A = B+C
D = D*E
```

(where the multiplication is by element).

### 39.6.2 Intrinsic functions

The following intrinsic functions are available for arrays:

- `Abs` creates the matrix of pointwise absolute values.
- `MaxLoc` returns the index of the maximum element.
- `MinLoc` returns the index of the minimum element.
- `MatMul` returns the matrix product of two matrices.
- `Dot_Product` returns the dot product of two arrays.
- `Transpose` returns the transpose of a matrix.
- `Cshift` rotates elements through an array.

Reduction operations on the array itself:

- **MaxVal** finds the maximum value in an array.
- **MinVal** finds the minimum value in an array.
- **Sum** returns the sum of all elements.
- **Product** return the product of all elements.

Reduction operations on a mask derived from the array:

- **All** finds if the mask is true for all elements
- **Any** finds if the mask is true for any element
- **Count** finds for how many elements the mask is true

- Functions such as **Sum** operate on a whole array by default.
- To restrict such a function to one subdimension add a keyword parameter **DIM**:

`s = Sum(A, DIM=1)`

where the keyword is optional.

- Likewise, the operation can be restricted to a **MASK**:

`s = Sum(A, MASK=B)`

#### Code:

```
// arrayf/normij.F90
! Summing in I and J
sums = Sum( A, dim=1 )
print *, "Row sums:"
print 10, sums

sums = Sum( A, dim=2 )
print *, "Column sums:"
print 10, sums
10 format( 4(i3,1x) )
```

#### Output:

```
[arrayf] rowcolsum:
Matrix:
 0   1   2   3
 4   5   6   7
 8   9  10  11
12  13  14  15
Row sums:
 6   22  38  54
Column sums:
24   28   32   36
```

**Exercise 39.5.** The 1-norm of a matrix is defined as the maximum of all sums of absolute values in any column:

$$\|A\|_1 = \max_j \sum_i |A_{ij}|$$

while the infinity-norm is defined as the maximum row sum:

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|$$

Compute these norms using array functions as much as possible, that is, try to avoid using loops.

For bonus points, write Fortran **Functions** that compute these norms.

**Exercise 39.6.** Compare implementations of the matrix-matrix product.

1. Write the regular  $i, j, k$  implementation, and store it as reference.
2. Use the **DOT\_PRODUCT** function, which eliminates the  $k$  index. How does the timing change?  
Print the maximum absolute distance between this and the reference result.
3. Use the **MATMUL** function. Same questions.
4. Bonus question: investigate the  $j, k, i$  and  $i, k, j$  variants. Write them both with array sections and individual array elements. Is there a difference in timing?

Does the optimization level make a difference in timing?

### 39.6.3 Restricting with `where`

If an array operation should not apply to all elements, you can specify the ones it applies to with a **where** statement.

```
where ( A<0 ) B = 0
```

Full form:

```
WHERE ( logical argument )
      sequence of array statements
ELSEWHERE
      sequence of array statements
END WHERE
```

### 39.6.4 Global condition tests

Reduction of a test on all array elements: **all**

```
REAL(8),dimension(N,N) :: A
LOGICAL :: positive,positive_row(N),positive_col(N)
positive = ALL( A>0 )
positive_row = ALL( A>0,1 )
positive_col = ALL( A>0,2 )
```

**Exercise 39.7.** Use array statements (that is, no loops) to fill a two-dimensional array  $A$  with random numbers between zero and one. Then fill two arrays  $A_{\text{big}}$  and  $A_{\text{small}}$  with the elements of  $A$  that are great than 0.5, or less than 0.5 respectively:

$$A_{\text{big}}(i,j) = \begin{cases} A(i,j) & \text{if } A(i,j) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$A_{\text{small}}(i,j) = \begin{cases} 0 & \text{if } A(i,j) \geq 0.5 \\ A(i,j) & \text{otherwise} \end{cases}$$

Using more array statements, add  $A_{\text{big}}$  and  $A_{\text{small}}$ , and test whether the sum is close enough to  $A$ .

Similar to **all**, there is a function **any** that tests if any array element satisfies the test.

```
if ( Any( Abs(A-B) >
```

## 39.7 Array operations

### 39.7.1 Loops without looping

In addition to ordinary do-loops, Fortran has mechanisms that save you typing, or can be more efficient in some circumstances.

#### 39.7.1.1 Slicing

If your loop assigns to an array from another array, you can use section notation:

```
a(:) = b(:)
c(1:n) = d(2:n+1)
```

#### 39.7.1.2 ‘forall’ keyword

The `forall` keyword also indicates an array assignment:

```
forall (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end forall
```

You can tell that this is for arrays only, because the loop index has to be part of the left-hand side of every assignment.

What happens if you apply `forall` to a statement with loop-carried dependencies? Consider first the traditional loops

<pre>// arrayf/forallshift.F90 A = [1,2,3,4,5] do i=1,4     A(i+1) = A(i) end do print '(5(i2x))',A</pre>	<pre>// arrayf/forallshift.F90 A = [1,2,3,4,5] do i=4,1,-1     A(i+1) = A(i) end do print '(5(i2x))',A</pre>
---	--

Can you predict the output? Now consider the following:

#### Code:

```
// arrayf/forallshift.F90
A = [1,2,3,4,5]
forall (i=1:4)
    A(i+1) = A(i)
end forall
print '(5(i2x))',A
```

#### Output:

[arrayf] **forallf:**

1 1 2 3 4

```
Code:
// arrayf forallshift.F90
A = [1,2,3,4,5]
do i=4,1,-1
    A(i+1) = A(i)
end do
print '(5(i2x))',A
```

```
Output:
[arrayf] forallb:
1 1 2 3 4
```

What does this tell you about the execution?

In other words, this mechanism is prone to misunderstanding and therefore now deprecated. It is not a parallel loop! For that, the following mechanism is preferred.

#### 39.7.1.3 Do concurrent

The *do concurrent* is a true do-loop. With the *concurrent* keyword the user specifies that the iterations of a loop are independent, and can therefore possibly be done in parallel:

```
do concurrent (i=1:n)
    a(i) = b(i)
    c(i) = d(i+1)
end do
```

(Do not use *for all*)

#### 39.7.2 Loops without dependencies

Here are some illustrations of simple array copying with the above mechanisms.

```
// slice/block.F90
do i=2,n
    counted(i) = 2*counting(i-1)
end do
```

Original	1	2	3	4	5	6	7	8	9	10
<b>Recursive</b>	0	2	4	6	8	10	12	14	16	18

```
// slice/block.F90
counted(2:n) = 2*counting(1:n-1)
```

Original	1	2	3	4	5	6	7	8	9	10
<b>Section</b>	0	2	4	6	8	10	12	14	16	18

```
// slice/block.F90
forall (i=2:n)
    counted(i) = 2*counting(i-1)
end forall
```

<i>Original</i>	1	2	3	4	5	6	7	8	9	10
<b>forall</b>	0	2	4	6	8	10	12	14	16	18

```
// slice/block.F90
do concurrent (i=2:n)
    counted(i) = 2*counting(i-1)
end do
```

<i>Original</i>	1	2	3	4	5	6	7	8	9	10
<i>Concurrent</i>	0	2	4	6	8	10	12	14	16	18

**Exercise 39.8.** Create arrays A, C of length 2N, and B of length N. Now implement

$$B_i = (A_{2i} + A_{2i+1})/2, \quad i = 1, \dots, N$$

and

$$C_i = A_{i/2}, \quad i = 1, \dots, 2N$$

using all four mechanisms. Make sure you get the same result every time.

### 39.7.3 Loops with dependencies

For parallel execution of a loop, all iterations have to be independent. This is not the case if the loop has a *recurrence*, and in this case, the ‘do concurrent’ mechanism is not appropriate. Here are the above four constructs, but applied to a loop with a dependence.

```
// slice/recur.F90
do i=2,n
    counting(i) = 2*counting(i-1)
end do
```

<i>Original</i>	1	2	3	4	5	6	7	8	9	10
<i>Recursive</i>	1	2	4	8	16	32	64	128	256	512

The slicing version of this:

```
// slice/recur.F90
counting(2:n) = 2*counting(1:n-1)
```

<i>Original</i>	1	2	3	4	5	6	7	8	9	10
<i>Section</i>	1	2	4	6	8	10	12	14	16	18

acts as if the right-hand side is saved in a temporary array, and subsequently assigned to the left-hand side.

Using ‘forall’ is equivalent to slicing:

```
// slice/recur.F90
forall (i=2:n)
```

```
    counting(i) = 2*counting(i-1)
end forall
```

*Original*      1    2    3    4    5    6    7    8    9    10  
**forall**        1    2    4    6    8    10   12   14   16   18

On the other hand, ‘do concurrent’ does not use temporaries, so it is more like the sequential version:

```
// slice/recur.F90
do concurrent (i=2:n)
    counting(i) = 2*counting(i-1)
end do
```

*Original*      1    2    3    4    5    6    7    8    9    10  
*Concurrent*    1    2    4    8    16   32   64   128   256   512

Note that the result does not have to be equal to the sequential execution: the compiler is free to rearrange the iterations any way it sees fit.

## 39.8 Review questions

**Exercise 39.9.** Let the following declarations be given, and assume that all arrays are properly initialized:

```
real                   :: x
real, dimension(10)   :: a, b
real, dimension(10,10) :: c, d
```

Comment on the following lines: are they legal, if so what do they do?

1.  $a = b$
2.  $a = x$
3.  $a(1:10) = c(1:10)$

How would you:

1. Set the second row of  $c$  to  $b$ ?
2. Set the second row of  $c$  to the elements of  $b$ , last-to-first?

# Chapter 40

## Pointers

Pointers in C/C++ are based on memory addresses; Fortran pointers on the other hand, are more abstract.

### 40.1 Basic pointer operations

Fortran pointers are a little like C or C++ pointers, and they are also different in many ways.

- Like C ‘star’ pointers, and unlike C++ ‘smart’ pointers, they can point at anything.
- Unlike C pointers, you have to declare that an object can be pointed at.
- Unlike any sort of pointer in C/C++, but like C++ references, they act as a sort of alias: there is no explicit dereferencing.

We will explore all this in detail.

Fortran pointers act like ‘aliases’: using a pointer variable is often the same as using the entity it points at. The difference with actually using the variable, is that you can decide what variable the pointer points at.

Fortran pointers are often automatically *dereferenced*: if you print a pointer you print the variable it references, not some representation of the pointer.

**Code:**

```
// pointerf/basicp.F90
real,target :: x
real,pointer :: point_at_real

x = 1.2
point_at_real => x
print *,point_at_real
```

**Output:**

```
[pointerf] basicp:
1.2000005
```

Pointers are defined in a variable declaration that specifies the type, with the `pointer` attribute. Examples: the definition

```
real,pointer :: point_at_real
```

declares a pointer that can point at a real variable. Without further specification, this pointer does not point at anything yet, so using it is undefined.

- You have to declare that a variable is point-able:

```
real,target :: x
```

- Declare a pointer:

```
real,pointer :: point_at_real
```

- Set the pointer with => notation (New! Note!):

```
point_at_real => x
```

Now using `point_at_real` is the same as using `x`.

```
print *,point_at_real ! will print the value of x
```

Pointers can not just point at anything: the thing pointed at needs to be declared as `target`

```
real,target :: x
```

and you use the `=>` operator to let a pointer point at a target:

```
point_at_real => x
```

If you use a pointer, for instance to print it

```
print *,point_at_real
```

it behaves as if you were using the value of what it points at.

#### Code:

```
// pointerf/realm.F90
real,target :: x,y
real,pointer :: that_real

x = 1.2
y = 2.4
that_real => x
print *,that_real
that_real => y
print *,that_real
y = x
print *,that_real
```

#### Output:

```
[pointerf] realm:
1.2000005
2.40000010
1.20000005
```

1. `that_real` points at `x`, so the value of `x` is printed.
2. `that_real` is reset to point at `y`, so its value is printed.
3. The value of `y` is changed, and since `that_real` still points at `y`, this changed value is printed.

## 40.2 Combining pointers

What happens if you point a pointer at another pointer? The concept of pointer-to-pointer from C/C++ does not exist: instead, you two pointers pointing at the same thing.

If you have two pointers

```
real,pointer :: point_at_real,also_point
```

you can make the target of the one to also be the target of the other:

```
point_at_real => x
also_point => point_at_real
```

Note that the second pointer is also assigned with the `=>` symbol. This is not a pointer to a pointer: it assigns the target of the right-hand side to be the target of the left-hand side.

What happens if you want to write `p2=>p1`  
but you write `p2=p1`?

The second one is legal, but has different meaning:

Assign underlying variables:

```
// pointerf/assignequals.F90
real,target :: x,y
real,pointer :: p1,p2
x = 1.2
p1 => x
p2 => y
p2 = p1 ! same as y=x
print *,p2 ! same as print y
```

Crash because `p2` pointer unassociated:

```
// pointerf/assignwrong.F90
real,target :: x
real,pointer :: p1,p2
x = 1.2
p1 => x
p2 = p1
print *,p2
```

**Exercise 40.1.** Write a routine that accepts an array and a pointer, and on return has that pointer pointing at the largest array element:

**Code:**

```
// pointerf/arpointf.F90
real,dimension(10),target :: array &
= [1.1, 2.2, 3.3, 4.4, 5.5, &
  9.9, 8.8, 7.7, 6.6, 0.0]
real,pointer :: biggest_element

print '(10f5.2)',array
call SetPointer(array,biggest_element)
print *, "Biggest element is",biggest_element
print *, "checking pointerhood:",&
  associated(biggest_element)
biggest_element = 0
print '(10f5.2)',array
```

**Output:**

[pointerf] arpointf:

```
1.10 2.20 3.30 4.40 5.50 9.90
  ↗8.80 7.70 6.60 0.00
Biggest element is 9.89999962
checking pointerhood: T
1.10 2.20 3.30 4.40 5.50 0.00
  ↗8.80 7.70 6.60 0.00
```

You can base this off the file `arpointf.F90` in the repository

### 40.3 Pointer status

A pointer can be in three states:

1. a pointer is undefined when it is first created,
2. it can be null, if explicitly set so,
3. or it can be associated if it has been pointed at something.

As a common sense strategy, do not worry about the undefined state: in the example in section 40.5 pointer are quickly made null.

- **Nullify**: zero a pointer
- **Associated**: test whether assigned

**Code:**

```
// pointerf/statusp.F90
real,target :: x
real,pointer :: realp

print *, "Pointer starts as not set"
if (.not.associated(realp)) &
    print *, "Pointer not associated"
x = 1.2
print *, "Set pointer"
realp => x
if (associated(realp)) &
    print *, "Pointer points"
print *, "Unset pointer"
nullify(realp)
if (.not.associated(realp)) &
    print *, "Pointer not associated"
```

**Output:**

[pointerf] statusp:

```
Pointer starts as not set
Pointer not associated
Set pointer
Pointer points
Unset pointer
Pointer not associated
```

You can also specifically test

- **associated**(*p, x*): whether the pointer is associated with the variable, or
- **associated**(*p1, p2*): whether two pointers are associated with the same target.

If you want a pointer to point at something,  
but you don't need a variable for that something:

**Code:**

```
// pointerf/allocptr.F90
Real,pointer :: x_ptr,y_ptr
allocate(x_ptr)
y_ptr => x_ptr
x_ptr = 6
print *,y_ptr
```

**Output:**

[pointerf] allocptr:

```
6.00000000
```

(Compare *make\_shared* in C++)

## 40.4 Pointers and arrays

You can set a pointer to an array element or a whole array.

```
real(8),dimension(10),target :: array
real(8),pointer           :: element_ptr
real(8),pointer,dimension(:) :: array_ptr

element_ptr => array(2)
array_ptr   => array
```

More surprising, you can set pointers to array sections:

```
array_ptr => array(2:)
array_ptr => array(1:size(array):2)
```

In case you're wondering, this does not create temporary arrays, but the compiler adds descriptions to the pointers, to translate code automatically to strided indexing.

You can use the `allocate` statement for pointers to arrays:

```
integer,pointer,dimension(:) :: array_point
Allocate( array_point(100) )
```

This is automatically deallocated when control leaves the scope. No memory leaks.

As an even more interesting example of pointers to array sections, let's consider the averaging operation

$$x_{i,j} = (x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1})/4.$$

We need pointers to the interior and its four offsets:

```
// pointerf/interior.F90
real(4),target,allocatable,dimension(:,:) :: grid
real(4),pointer,dimension(:,:,:) :: interior,left,right,up,down

Allocate( grid(N,N) )
/* ... */
interior => grid(2:N-1,2:N-1)
up      => grid(1:N-2,2:N-1)
down    => grid(3:N,2:N-1)
left    => grid(2:N-1,1:N-3)
right   => grid(2:N-1,3:N)
```

The averaging operation is then an array statement:

```
// pointerf/interior.F90
interior = ( up+down+left+right )/4
```

## 40.5 Example: linked lists

For pictures of linked lists, see section 66.1.2.

- Linear data structure
- more flexible than array for insertion / deletion
- ... but slower in access

One of the standard examples of using pointers is the *linked list*. This is a dynamic one-dimensional structure that is more flexible than an array. Dynamically extending an array would require re-allocation, while in a list an element can be inserted.

**Exercise 40.2.** Using a linked list may be more flexible than using an array. On the other hand, accessing an element in a linked list is more expensive, both absolutely and as order-of-magnitude in the size of the list.

Make this argument precise.

#### 40.5.1 Type definitions

A list is based on a simple data structure, a node, which contains a value field and a pointer to another node. The list data structure itself only contains a pointer to the first node in the list.

- Node: value field, and pointer to next node.
  - List: pointer to head node.
- ```
// pointerf/listfappendalloc.F90
type node
    integer :: value
    type(node),pointer :: next
end type node

type list
    type(node),pointer :: head
end type list
```

By way of example, we create a dynamic list of integers, sorted by size. To maintain the sortedness, we need to append or insert nodes, as required.

Our main program will create three nodes, and append them to the end of the list:

|                                                                                                                                                                                                                                                                                                                                                   |                                                                                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| <b>Code:</b><br><pre>// pointerf/listfappendalloc.F90 integer,parameter :: listsize=7 type(list) :: the_list integer,dimension(listsize) :: inputs = &amp; [ 62, 75, 51, 12, 14, 15, 16 ] integer :: input,input_value  nullify(the_list%head) do input=1,listsize   input_value = inputs(input)   call attach(the_list,input_value) end do</pre> | <b>Output:</b><br><b>[pointerf] listappend:</b><br>List: [ 62, 75, 51, 12, 14, 15, 16, ] |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|

### 40.5.2 Attach a node at the end

First we write a routine `attach` that takes a node pointer and attaches it to the end of a list, without any sorting.

```
// pointerf/listfappendalloc.F90
subroutine attach( the_list,new_value )
  implicit none
  ! parameters
  type(list),intent(inout) :: the_list
  integer,intent(in) :: new_value
```

We distinguish two cases: when the list is empty, and when it is not. Initially, the list is empty, meaning that the ‘head’ pointer is un-associated. The first time we add an element to the list, we assign the node as the head of the list:

```
// pointerf/listfappendalloc.F90
! if the list has no head node, attached the new node
if (.not.associated(the_list%head)) then
  allocate( the_list%head )
  the_list%head%value = new_value
else
  call node_attach( the_list%head,new_value )
end if
```

New element attached at the end.

```
// pointerf/listfappendalloc.F90
recursive subroutine node_attach( the_node,new_value )
/* ... */
if ( .not. associated(the_node%next) ) then
  allocate( the_node%next )
  the_node%next%value = new_value
else
```

```
    call node_attach( the_node%next,new_value )
end if
```

**Exercise 40.3.** Take the recursive code for attaching an element, and turn it into an iterative version, that is, use a `while` loop that goes down the list till the end.

You may do the whole thing in the `attach` routine for the list head.

#### 40.5.3 Insert a node in sort order

If we want to keep a list sorted, we need in many cases to insert the new node at a location short of the end of the list. This means that instead of iterating to the end, we iterate to the first node that the new one needs to be attached to.

Almost the same as before, but now keep the list sorted:

**Code:**

```
// pointerf/listfinsertalloc.F90
do in=1,listsize
  in_value = inputs(in)
  call insert(the_list,in_value)
  call print(the_list)
end do
```

**Output:**

```
[pointerf] listinsert:
List: [ 62 ]
List: [ 62 75 ]
List: [ 51 62 75 ]
List: [ 12 51 62 75 ]
List: [ 12 14 51 62 75 ]
List: [ 12 14 15 51 62 75
      ↵ ]
List: [ 12 14 15 16 51 62
      ↵ 75 ]
```

**Exercise 40.4.** Copy the `attach` routine to `insert`, and modify it so that inserting a value will keep the list ordered.

You can base this off the file `listfappendalloc.F90` in the repository

**Exercise 40.5.** Modify your code from exercise 40.4 so that the new node is not allocated in the main program.

Instead, pass only the integer argument, and use `allocate` to create a new node when needed.

**Exercise 40.6.** Write a `print` function for the linked list.

For the simplest solution, print each element on a line by itself.

More sophisticated: use the `write` function and the `advance` keyword:

```
write(*,'(i1,"")',advance="no") current%value
```

**Exercise 40.7.** Write a `length` function for the linked list.

Try it both with a loop, and recursively.

# Chapter 41

## Input/output

### 41.1 Types of I/O

Fortran can deal with input/output in ASCII format, which is called *formatted I/O*, and binary, or *unformatted I/O*. Formatted I/O can use default formatting, but you can specify detailed formatting instructions, in the I/O statement itself, or separately in a `Format` statement.

Fortran I/O can be described as *list-directed I/O*: both input and output commands take a list of item, possibly with formatting specified.

- `Print` simple output to terminal
- `Write` output to terminal or file ('unit')
- `Read` input from terminal or file
- `Open, Close` for files and streams
- `Format` format specification that can be used in multiple statements.

- Formatted: ASCII output. This is good for reporting, but not for numeric data storage.
- Unformatted: binary output. Great for further processing of output data.
- Beware: binary data is machine-dependent. Use `hdf5` for portable binary.

### 41.2 Print to terminal

The simplest command for outputting data is `print`.

```
print *, "The result is", result
```

In its easiest form, you use the star to indicate that you don't care about formatting; after the comma you can then have any number of comma-separated strings and variables.

#### 41.2.1 Print on one line

The statement

```
print *, item1, item2, item3
```

will print the items on one line, as far as the line length allows.

Parameterized printing with an *implicit do loop*:

```
print *, ( i*i, i=1, n)
```

All values will be printed on the same line.

### 41.2.2 Printing arrays

If you print a variable that is an array, all its elements will be printed, in *column-major* ordering if the array is multi-dimensional.

You can also control the printing of an array by using an *implicit do loop*:

```
print *, ( A(i), i=1, n)
```

## 41.3 Formatted I/O

The default formatting uses quite a few positions for what can be small numbers. To indicate explicitly the formatting, for instance limiting the number of positions used for a number, or the whole and fractional part of a real number, you can use a format string.

For instance, you can use a letter followed by a digit to control the formatting width:

|                                                                                                                                                |                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------|
| <b>Code:</b>                                                                                                                                   | <b>Output:</b>                                    |
| <pre>// iof/format.F90 i = 56 print *, i print '(i4)', i print '(i2)', i print '(i1)', i i = i*i print '("fit &lt;", i0, "&gt; ted")', i</pre> | <pre>[iof] i4: 56 56 * fit &lt;3136&gt; ted</pre> |

(In the last line, the format specifier was not wide enough for the data, so an *asterisk* was used as output.)

Let's approach this semi-systematically.

### 41.3.1 Format letters

#### 41.3.1.1 Integers

Integers can be set with *in*.

- If  $n > 0$ , that many positions are used with the number right aligned; except
- if the number does not fit in  $n$  positions, it is rendered as asterisks.
- To use precisely the required number of positions, use *i0*.

|                                                                                                                                                              |                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------|
| <b>Code:</b><br><pre>// iof/format.F90 i = 56 print *, i print '(i4)', i print '(i2)', i print '(i1)', i i = i*i print '("fit &lt;",i0,"&gt; ted")', i</pre> | <b>Output:</b><br><pre>[iof] i4: 56 56 * fit &lt;3136&gt; ted</pre> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------|

### 41.3.1.2 Strings

Strings can be handled two ways:

1. They can be literally part of the format string:

```
print '(i2,"--",i2)', m, n
```

2. The can be formatted with the `a`*n* specifier:

```
print '(a5,2)', somestring, someint
```

3. To use precisely the required number of positions, use `a`

```
print '(a,i0,a)', str1,int2,str3
```

### 41.3.1.3 Floating point

- ‘`fm.n`’ specifies a fixed point representation of a real number, with *m* total positions (including the decimal point) and *n* digits in the fractional part.
- ‘`em.n`’ Exponent representation.

### 41.3.1.4 Other

```
x : one space
x5 : five spaces
```

```
b : binary
o : octal
z : hex
```

## 41.3.2 Repeating and grouping

If you want to display items of the same type, you can use a repeat count:

## 41. Input/output

---

**Code:**

```
// iof/format.F90
i = 12; j = 34
print '(2i4)', i, j
print '(2i2)', i, j
```

**Output:**

```
[iof] ii:
12 34
1234
```

You can mix variables of different types, as well as literal string, by separating them with commas. And you can group them with parentheses, and put a repeat count on those groups again:

**Code:**

```
// iof/format.F90
i = 23; j = 45; k = 67
print '(i2,1x,i2)', i, j
print ("Numbers:", 3(1x,i2,".")), i, j, k
```

**Output:**

```
[iof] ij:
23 45
Numbers: 23. 45. 67.
```

Putting a number in front of a single specifier indicates that it is to be repeated.

If the data takes more positions than the format specifier allows, a string of asterisks is printed:

**Code:**

```
// fio/asterisk.F90
do ipower=1,5
    print '(i3)', number
    number = 10*number
end do
```

**Output:**

```
[fio] asterisk:
1
10
100
***
***
```

If you find yourself using the same format a number of times, you can give it a *label*:

```
print 10,"result:",x,y,z
10 format('(a6,3f5.3)')
```

<https://www.obliquity.com/computer/fortran/format.html>

```
print '( 3i4 )', i1,i2,i3
print '( 3(i2,":",f7.4) )', i1,r1,i2,r2,i3,r2
```

- If *abc* is a format string, then 10(*abc*) gives 10 repetitions. There is no line break.
- If there is more data than specified in the format, the format is reused in a new print statement. This causes line breaks.
- The / (slash) specifier causes a line break.
- There may be a 80 character limit on output lines.

**Exercise 41.1.** Use formatted I/O to print the number 0 ··· 99 as follows:

```

0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99

```

## 41.4 File and stream I/O

If you want to send output anywhere else than the terminal screen, you need the `write` statement, which looks like:

```
write (unit,format) data
```

where `format` and `data` are as described above. The new element is the `unit`, which is a numerical indication of an output device, such as a file.

### 41.4.1 Units

For file I/O you write to a unit number, which is associated with a file through an `open` statement.

After you are done with the file, you `Close` it.

```
Open(11)
```

will result in a file with a name typically `fort11`. To give it a name of your choosing:

```
Open(11,FILE="filename")
```

Many other options for error handling, new vs old file, etc.

After this, a `Write` statement can refer to the unit:

```
Write (11,fmt) data
```

Again options for errors and such.

### 41.4.2 Other write options

By default, each `Write` statement, like a `Print` statement, writes to a new line (or ‘record’ in Fortran terminology). To prevent this,

```
write(unit,fmt,ADVANCE="no") data
```

will not issue a newline.

## 41.5 Conversion to/from string

Above, you saw the commands `Print`, `Write`, and `Read` in the context of output to/from terminal and file. However, there is a second type of use for `Write` and `Read` for string handling, namely conversion between strings and numerical quantities.

### 41.5.0.1 String to numeric

To convert a string to numerical quantities, perform a `Read` operation:

```
Read( some_string, some_format ) bunch, of, quantities
```

Example:

|                                                                                                                                                                                                                                                                                                |                                                                                       |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <b>Code:</b><br><pre>// stringf/readwrite.F90 character(len=8) :: date integer :: year,month,day date = "20221027" read( date,'( i4,i2,i2 )' ) &amp;      year,month,day /* ... */ print *, "Date:",date print '( "Year=",i4, ", mo=",i2, ", day=",i2 )',      &amp;      year,month,day</pre> | <b>Output:</b><br><b>[stringf] date:</b><br>Date:20221027<br>Year=2022, mo=10, day=27 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|

### 41.5.0.2 Numeric to string

Conversely, to construct a string from some quantities you would perform a `Write` operation:

```
Write( some_string, some_format ) bunch, of, quantities
```

|                                                                                                                                                                                                            |                                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| <b>Code:</b><br><pre>// stringf/readwrite.F90 character(len=10) :: longdate /* ... */ write( longdate,&amp;       '( i4,"/",i2,"/",i2 )' &amp;       ) year,month,day print *, "Long date:",longdate</pre> | <b>Output:</b><br><b>[stringf] slash:</b><br>Long date:2022/10/27 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|

## 41.6 Unformatted output

So far we have looked at ASCII output, which is nice to look at for a human , but is not the right medium to communicate data to another program.

- ASCII output requires time-consuming conversion.
- ASCII rendering leads to loss of precision.

Therefore, if you want to output data that is later to be read by a program, it is best to use *binary output* or *unformatted output*, sometimes also called *raw output*.

Indicated by lack of format specification:

```
write (*) data
```

Note: may not be portable between machines.

## 41.7 Print to printer

In Fortran standards before Fortran2003, column 1 of the output had a special meaning, corresponding to *line printer* tractor control. Notoriously, having a character here would move to a new page. While this feature has been removed from the standard, you may still see a black first column in your output, without specifying such.



## Chapter 42

### Leftover topics

#### 42.1 Interfaces

If you want to use a procedure in your main program, the compiler needs to know the signature of the procedure: how many arguments, of what type, and with what intent. You have seen how the `contains` clause can be used for this purpose if the procedure resides in the same file as the main program.

If the procedure is in a separate file, the compiler does not see definition and usage in one go. To allow the compiler to do checking on proper usage, we can use an `interface` block. This is placed at the calling site, declaring the signature of the procedure.

##### Main program:

```
// funcf/interface.F90
interface
    function f(x,y)
        real*8 :: f
        real*8,intent(in) :: x,y
    end function f
end interface

real*8 :: in1=1.5, in2=2.6, result

result = f(in1,in2)
```

##### Procedure:

```
// funcf/interfunc.F90
function f(x,y)
    implicit none
    real*8 :: f
    real*8,intent(in) :: x,y
```

The `interface` block is not required (an older `external` mechanism exists for functions), but is recommended. It is required if the function takes function arguments.

##### 42.1.1 Polymorphism

The `interface` block can be used to define a generic function:

```
interface f
function f1(.....)
function f2(.....)
end interface f
```

where  $f_1, f_2$  are functions that can be distinguished by their argument types. The generic function  $f$  then becomes either  $f_1$  or  $f_2$  depending on what type of argument it is called with.

## 42.2 Random numbers

In this section we briefly discuss the Fortran *random number generator*. The basic mechanism is through the library subroutine `random_number`, which has a single argument of type `REAL` with `INTENT(OUT)`:

```
real(4) :: randomfraction
call random_number(randomfraction)
```

The result is a random number from the uniform distribution on  $[0, 1]$ .

Setting the *random seed* is slightly convoluted. The amount of storage needed to store the seed can be processor and implementation-dependent, so the routine `random_seed` can have three types of named argument, exactly one of which can be specified at any one time. The keyword can be:

- `SIZE` for querying the size of the seed;
- `PUT` for setting the seed; and
- `GET` for querying the seed.

A typical fragment for setting the seed would be:

```
integer :: seedsize
integer,dimension(:),allocatable :: seed

call random_seed(size=seedsize)
allocate(seed(seedsize))
seed(:) = ! your integer seed here
call random_seed(put=seed)
```

## 42.3 Timing

Timing is done with the `system_clock` routine.

- This call gives an integer, counting clock ticks.
- To convert to seconds, it can also tell you how many ticks per second it has: its *timer resolution*.

```
integer :: clockrate,clock_start,clock_end
call system_clock(count_rate=clockrate)
print *, "Ticks per second:",clockrate

call system_clock(clock_start)
! code
call system_clock(clock_end)
print *, "Time:",(clock_end-clock_start)/REAL(clockrate)
```

## 42.4 Fortran standards

- The first Fortran standard was just called ‘Fortran’.
  - Fortran4 was a popular next standard.
  - Fortran66 was also common. It was very much based on `goto` statements, because there were no block structures.
  - Fortran77 was a more structured language, containing the modern `do` and `if` block statements. However, memory management was still completely static and recursive procedures didn’t exist.
  - Fortran88 didn’t happen in time to justify the name, and even calling it Fortran8X didn’t help: it became Fortran90. This standard introduced
    - Modules, which made `common` blocks no longer needed.
    - the `implicit none` specification,
    - dynamic memory allocation.
    - recursion.
- Fortran95 was a clarification of Fortran90.
- Fortran2003 (and its refinement Fortran2008) introduced:
    - Object-orientation.
    - This is also when Co-array Fortran (CAF) became part of the language.
  - Fortran2018 introduced sub-modules, more parallelism, more C-interoperability.



## Chapter 43

### Fortran review questions

#### 43.1 Fortran versus C++

**Exercise 43.1.** For each of C++, Fortran, Python:

- Give an example of an application or application area that the language is suited for, and
- Give an example of an application or application area that the language is not so suited for.

#### 43.2 Basics

**Exercise 43.2.**

- What does the `Parameter` keyword do? Give an example where you would use it.
- Why would you use a `Module`?
- What is the use of the `Intent` keyword?

#### 43.3 Arrays

**Exercise 43.3.** You are looking at historical temperature data, say a table of the high and low temperature at January 1st of every year between 1920 and now, so that is 100 years.

Your program accepts data as follows:

```
Integer :: year, high, low  
!! code omitted  
read *,year,high,low
```

where the temperatures are rounded to the closest degree (Centigrade of Fahrenheit is up to you.)

Consider two scenarios. For both, give the lines of code for 1. the array in which you store the data, 2. the statement that inserts the values into the array.

- Store the raw temperature data.

- Suppose you are interested in knowing how often certain high/low temperatures occurred. For instance, ‘how many years had a high temperature of 32F /0 C’.

## 43.4 Subprograms

**Exercise 43.4.** Write the missing procedure pos\_input that

- reads a number from the user
- returns it
- and returns whether the number is positive

in such a way to make this code work:

**Code:**

```
// funcf/looppos.F90
program looppos
    implicit none
    real(4) :: userinput
    do while (pos_input(userinput))
        print &
            ('Positive input:', f7.3), &
            userinput
    end do
    print &
        ('Nonpositive input:', f7.3), &
        userinput
    /* ... */
end program looppos
```

**Output:**

[funcf] looppos:

Running with the following  
→inputs:

5  
1  
-7.3

/bin/sh: ./looppos: No such  
→file or directory  
make[2]: \*\*\* [run\_looppos]  
→Error 127

Give the function parameter(s) the right **Intent** directive.

Hint: is pos\_input a SUBROUTINE or FUNCTION? If the latter, what is the type of the function result? How many parameters does it have otherwise? Where does the variable user\_input get its value? So what is the type of the parameter(s) of the function?

**PART IV**

**EXERCISES AND PROJECTS**



## Chapter 44

### Style guide for project submissions

*The purpose of computing is insight, not numbers. (Richard Hamming)*

Your project writeup is equally important as the code. Here are some common-sense guidelines for a good writeup. However, not all parts may apply to your project. Use your good judgment.

#### 44.1 General approach

As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

Turn in a writeup in pdf form (Word and text documents are not acceptable) that was generated from a text processing program such (preferably) L<sup>A</sup>T<sub>E</sub>X. For a tutorial, see Tutorials book [11], section 15.

#### 44.2 Style

Your report should obey the rules of proper English.

- Observing correct spelling and grammar goes without saying.
- Use full sentences.
- Try to avoid verbiage that is disparaging or otherwise inadvisable. The *Google developer documentation style guide* [16] is a great resource.

#### 44.3 Structure of your writeup

Consider this project writeup an opportunity to practice writing a scientific article.

Start with the obvious stuff.

- Your writeup should have a title. Not ‘Project’, but something like ‘Simulation of Chronosynclastic Enfundibula’.
- Author and contact information. This differs per case. Here it is: your name, EID, TACC username, and email.
- Introductory section that is extremely high level: what is the problem, what did you do, what did you find.

- Conclusion: what do your findings mean, what are limitations, opportunities for future extensions.
- Bibliography.

#### 44.3.1 Introduction

The reader of your document need not be familiar with the project description, or even the problem it addresses. Indicate what the problem is, give theoretical background if appropriate, possibly sketch a historic background, and describe in global terms how you set out to solve the problem, as well as a brief statement of your findings.

#### 44.3.2 Detailed presentation

See section 44.5 below.

#### 44.3.3 Discussion and summary

Separate the detailed presentation from a discussion at the end: you need to have a short final section that summarizes your work and findings. You can also discuss possible extensions of your work to cases not covered.

### 44.4 Experiments

You should not expect your program to run once and give you a final answer to your research question. Ask yourself: what parameters can be varied, and then vary them! This allows you to generate graphs or multi-dimensional plots.

If you vary a parameter, think about what granularity you use. Do ten data points suffice, or do you get insight from using 10,000?

Above all: computers are very fast, they do a billion operations per second. So don't be shy in using long program runs. Your program is not a calculator where a press on the button immediately gives the answer: you should expect program runs to take seconds, maybe minutes.

### 44.5 Detailed presentation of your work

The detailed presentation of your work is as combination of code fragments, tables, graphs, and a description of these.

#### 44.5.1 Presentation of numerical results

You can present results as graphs/diagrams or tables. The choice depends on factors such as how many data points you have, and whether there is an obvious relation to be seen in a graph.

Graphs can be generated any number of ways. Kudos if you can figure out the  $\text{\LaTeX}$  `tikz` package, but Matlab or Excel are acceptable too. No screenshots though.

Number your graphs/tables and refer to the numbering in the text. Give the graph a clear label and label the axes.

#### 44.5.2 Code

Your report should describe in a global manner the algorithms you developed, and you should include relevant code snippets. If you want to include full listings, relegate that to an appendix: code snippets in the text should only be used to illustrate especially salient points.

Do not use screen shots of your code: at the very least use a monospace font such as the `verbatim` environment, but using the `listings` package (used in this book) is very much recommended.



# Chapter 45

## Prime numbers

In this chapter you will do a number of exercises regarding prime numbers that build on each other. Each section lists the required prerequisites. Conversely, the exercises here are also referenced from the earlier chapters.

### 45.1 Arithmetic

*Before doing this section, make sure you study section 4.5.*

**Exercise 45.1.** Read two numbers and print out their modulus. The modulus operator is `x%y`.

- Can you also compute the modulus without the operator?
- What do you get for negative inputs, in both cases?
- Assign all your results to a variable before outputting them.

### 45.2 Conditionals

*Before doing this section, make sure you study section 5.1.*

**Exercise 45.2.** Read two numbers and print a message stating whether the second is a divisor of the first:

```
Code:
// primes/divisiontest.cpp
int number, divisor;
bool is_a_divisor;
/* ... */
if (
/* ... */
) {
    cout << "Yes, " << divisor
    << " is a divisor of "
    << number << '\n';
} else {
    cout << "No, " << divisor
    << " is not a divisor of "
    << number << '\n';
}
```

```
Output:
[primes] division:
( echo 6 ; echo 2 ) |
    ↪divisiontest
Enter a number:
Enter a trial divisor:
Indeed, 2 is a divisor of 6

( echo 9 ; echo 2 ) |
    ↪divisiontest
Enter a number:
Enter a trial divisor:
No, 2 is not a divisor of 9
```

### 45.3 Looping

*Before doing this section, make sure you study section 6.1.*

**Exercise 45.3.** Read an integer and set a boolean variable to determine whether it is prime by testing for the smaller numbers if they divide that number.

Print a final message

Your number is prime

or

Your number is not prime: it is divisible by ....

where you report just one found factor.

Printing a message to the screen is hardly ever the point of a serious program. In the previous exercise, let's therefore assume that the fact of primeness (or non-primeness) of a number will be used in the rest of the program. So you want to store this conclusion.

**Exercise 45.4.** Rewrite the previous exercise with a boolean variable to represent the primeness of the input number.

**Exercise 45.5.** Read in an integer  $r$ . If it is prime, print a message saying so. If it is not prime, find integers  $p \leq q$  so that  $r = p \cdot q$  and so that  $p$  and  $q$  are as close together as possible. For instance, for  $r = 30$  you should print out  $5, 6$ , rather than  $3, 10$ . You are allowed to use the function `sqrt`.

### 45.4 Functions

*Before doing this section, make sure you study section ??.*

chapter]ch:function

Above you wrote several lines of code to test whether a number was prime. Now we'll turn this code into a function.

**Exercise 45.6.** Write a function `is_prime` that has an integer parameter, and returns a boolean corresponding to whether the parameter was prime.

```
int main() {
    bool isprime;
    isprime = is_prime(13);
```

Write a main program that reads the number in, and prints the value of the boolean.

Does your function have one or two `return` statements? Can you imagine what the other possibility looks like? Do you have an argument for or against it?

## 45.5 While loops

*Before doing this section, make sure you study section 6.3.*

**Exercise 45.7.** Take your prime number testing function `is_prime`, and use it to write a program that prints multiple primes:

- Read an integer `how_many` from the input, indicating how many (successive) prime numbers should be printed.
- Print that many successive primes, each on a separate line.
- (Hint: keep a variable `number_of_primes_found` that is increased whenever a new prime is found.)

## 45.6 Classes and objects

*Before doing this section, make sure you study section 9.1.*

**Exercise 45.8.** Write a class `primegenerator` that contains:

- Methods `number_of_primes_found` and `nextprime`;
- Also write a function `isprime` that does not need to be in the class.

Your main program should look as follows:

```
// primes/6primesbyclass.cpp
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found() < nprimes) {
    int number = sequence.nextprime();
    cout << "Number " << number << " is prime" << '\n';
}
```

In the previous exercise you defined the `primegenerator` class, and you made one object of that class:

```
primegenerator sequence;
```

But you can make multiple generators, that all have their own internal data and are therefore independent of each other. The following example illustrates this.

The *Goldbach conjecture* says that every even number, from 4 on, is the sum of two primes  $p + q$ .

$$\forall_{\text{even}(n)} \exists_{p,q} : \text{prime}(p) \wedge \text{prime}(q) \wedge n = p + q.$$

**Exercise 45.9.** Write a program to test the Goldbach conjecture for the even numbers up to a bound that you read in.

First formulate the quantor structure of this statement, then translate that top-down to code, using the generator you developed above.

1. Make an outer loop over the even numbers  $e$ .
2. For each  $e$ , generate all primes  $p$ .
3. From  $p + q = e$ , it follows that  $q = e - p$  is prime: test if that  $q$  is prime.

For each even number  $e$  then print  $e, p, q$ , for instance:

The number 10 is 3+7

If multiple possibilities exist, only print the first one you find.

An interesting corollary of the Goldbach conjecture is that each prime (start at 5) is equidistant between two other primes.

The *Goldbach conjecture* says that every even number  $2n$  (starting at 4), is the sum of two primes  $p + q$ :

$$2n = p + q.$$

Equivalently, every number  $n$  is equidistant from two primes:

$$n = \frac{p + q}{2} \quad \text{or} \quad q - n = n - p.$$

In particular this holds for each prime number:

$$\forall_{r \text{ prime}} \exists_{p,q \text{ prime}} : r = (p + q)/2 \text{ is prime.}$$

We now have the statement that each prime number is the average of two other prime numbers.

**Exercise 45.10.**

Write a program that tests this. You need at least one loop that tests all primes  $r$ ; for each  $r$  you then need to find the primes  $p, q$  that are equidistant to it.

Use your prime generator. Do you use two generators for this, or is one enough? Do you need three, for  $p, q, r$ ?

For each  $r$  value, when the program finds the  $p, q$  values, print the  $p, q, r$  triple and move on to the next  $r$ .

### 45.6.1 Optimization

**Exercise 45.11.** In the previous Goldbach exercise you had a prime number generator in a loop, meaning that primes got recalculated a number of times.

Optimize your prime number generator so that it remembers numbers already requested.

Hint: have a `static` vector.

### 45.6.2 Exceptions

*Before doing this section, make sure you study section 23.2.*

**Exercise 45.12.** Revisit the prime generator class (exercise 45.8) and let it throw an exception once the candidate number is too large. (You can hardwire this maximum, or use a limit; section 24.2.)

**Code:**

```
// primes/genx.cpp
try {
    do {
        auto cur = primes.nextprime();
        cout << cur << '\n';
    } while (true);
} catch ( string s ) {
    cout << s << '\n';
}
```

**Output:**

```
[primes] genx:
9931
9941
9949
9967
9973
Reached max int
```

### 45.6.3 Prime number decomposition

*Before doing this section, make sure you study section 24.3.1.*

Design a class `Integer` which stores its value as its prime number decomposition. For instance,

$$180 = 2^2 \cdot 3^3 \cdot 5 \quad \Rightarrow \quad [2:2, 3:2, 5:1]$$

You can implement this decomposition itself as a vector, (the  $i$ -th location stores the exponent of the  $i$ -th prime) but let's use a `map` instead.

**Exercise 45.13.** Write a constructor of an `Integer` from an `int`, and methods `as_int / as_string` that convert the decomposition back to something classical. Start by assuming that each prime factor appears only once.

**Code:**

```
// primes/decomposition.cpp
Integer i2(2);
cout << i2.as_string() << ":" << i2.as_int() << '\n';

Integer i6(6);
cout << i6.as_string() << ":" << i6.as_int() << '\n';
```

**Output:**  
**[primes] decomposition26:**  
 $2^1 : 2$   
 $2^1 3^1 : 6$

**Exercise 45.14.** Extend the previous exercise to having multiplicity  $> 1$  for the prime factors.

**Code:**

```
// primes/decomposition.cpp
Integer i180(180);
cout << i180.as_string() << ":" << i180.as_int() << '\n';
```

**Output:**  
**[primes] decomposition180:**  
 $2^2 3^2 5^1 : 180$

Implement addition and multiplication for *Integers*.

Implement a class *Rational* for rational numbers, which are implemented as two *Integer* objects. This class should have methods for addition and multiplication. Write these through operator overloading if you've learned this.

Make sure you always divide out common factors in the numerator and denominator.

## 45.7 Ranges

*Before doing this section, make sure you study section 14.1.*

**Exercise 45.15.** Write a range-based code that tests

$$\forall \text{prime } p : \exists \text{prime } q : q > p$$

**Exercise 45.16.** Rewrite exercise 45.10, using only range expressions, and no loops.

**Exercise 45.17.** In the above Goldbach exercises you probably needed two prime number sequences, that, however, did not start at the same number. Can you make it so that your code reads

```
all_of( primes_from(5) /* et cetera */
```

## 45.8 Other

The following exercise requires *std::optional*, which you can learn about in section 24.6.2.

**Exercise 45.18.** Write a function `first_factor` that optionally returns the smallest factor of a given input.

```
// primes/optfactor.cpp
auto factor = first_factor(number);
if (factor.has_value())
    cout << "Found factor: " << factor.value() << '\n';
else
    cout << "Prime number\n";
```

## 45.9 Eratosthenes sieve

The Eratosthenes sieve is an algorithm for prime numbers that step-by-step filters out the multiples of any prime it finds.

1. Start with the integers from 2: 2, 3, 4, 5, 6, ...
2. The first number, 2, is a prime: record it and remove all multiples, giving

3, 5, 7, 9, 11, 13, 15, 17...

3. The first remaining number, 3, is a prime: record it and remove all multiples, giving

5, 7, 11, 13, 17, 19, 23, 25, 29...

4. The first remaining number, 5, is a prime: record it and remove all multiples, giving

7, 11, 13, 17, 19, 23, 29, ...

### 45.9.1 Arrays implementation

The sieve can be implemented with an array that stores all integers.

**Exercise 45.19.** Read in an integer that denotes the largest number you want to test. Make an array of integers that long. Set the elements to the successive integers. Apply the sieve algorithm to find the prime numbers.

### 45.9.2 Streams implementation

The disadvantage of using an array is that we need to allocate an array. What's more, the size is determined by how many integers we are going to test, not how many prime numbers we want to generate. We are going to take the idea above of having a generator object, and apply that to the sieve algorithm: we will now have multiple generator objects, each taking the previous as input and erasing certain multiples from it.

**Exercise 45.20.** Write a `stream` class that generates integers and use it through a pointer.

**Code:**

```
// sieve/ints.cpp
for (int i=0; i<7; ++i)
    cout << "Next int: "
    << the_ints->next() << '\n';
```

**Output:**

```
[sieve] ints:
Next int: 2
Next int: 3
Next int: 4
Next int: 5
Next int: 6
Next int: 7
Next int: 8
```

Next, we need a stream that takes another stream as input, and filters out values from it.

**Exercise 45.21.** Write a class *filtered\_stream* with a constructor

```
filtered_stream(int filter, shared_ptr<stream> input);
```

that

1. Implements *next*, giving filtered values,
2. by calling the *next* method of the input stream and filtering out values.

**Code:**

```
// sieve/odds.cpp
auto integers =
    make_shared<stream>();
auto odds =
    shared_ptr<stream>
    ( new filtered_stream(2, integers) );
for (int step=0; step<5; ++step)
    cout << "next odd: "
    << odds->next() << '\n';
```

**Output:**

```
[sieve] odds:
next odd: 3
next odd: 5
next odd: 7
next odd: 9
next odd: 11
```

Now you can implement the Eratosthenes sieve by making a *filtered\_stream* for each prime number.

**Exercise 45.22.** Write a program that generates prime numbers as follows.

- Maintain a *current* stream, that is initially the stream of prime numbers.
- Repeatedly:
  - Record the first item from the current stream, which is a new prime number;
  - and set *current* to a new stream that takes *current* as input, filtering out multiples of the prime number just found.

## 45.10 Range implementation

*Before doing this section, make sure you study section 14.6.3.*

If we write the prime number definition

$$\begin{aligned} D(n, d) &\equiv n|d = 0 \\ P(n) &\equiv \forall_{d \leq \sqrt{N}}: \neg D(n, d) \end{aligned}$$

we see that this involves two streams that we iterate over:

1. First there is the set of all  $d$  such that  $d^2 \leq n$ ; then
2. We have the set of booleans testing whether these  $d$  values are divisors.

**Exercise 45.23.** Use the `iota` range view to generate all integers from 2 to infinity, and find a range view that cuts off the sequence at the last possible divisor.

Then use the `all_of` or `any_of` ranged algorithms to test whether any of these potential divisors are actually a divisor, and therefore whether or not your number is prime.

**Exercise 45.24.** Use the `filter` view to filter from an `iota` view those elements that are prime.

**Exercise 45.25.** Make a `primes` class that can be ranged:

**Code:**

```
// primes/rangeclass.cpp
primegenerator allprimes;
for ( auto p : allprimes ) {
    cout << p << ", ";
    if (p>100) break;
}
cout << '\n';
```

**Output:**

```
[primes] rangeclass:
2, 3, 5, 7, 11, 13, 17, 19, 23,
→29, 31, 37, 41, 43, 47,
→53, 59, 61, 67, 71, 73,
→79, 83, 89, 97, 101,
```

## 45.11 User-friendliness

Use the `cxxopts` package (section 63.2) to add commandline options to some primality programs.

**Exercise 45.26.** Take your old prime number testing program, and add commandline options:

- the `-h` option should print out usage information;
- specifying a single int `--test 1001` should print out all primes under that number;
- specifying a set of ints `--tests 57,125,1001` should test primeness for those.



## Chapter 46

### Geometry

In this set of exercises you will write a small ‘geometry’ package: code that manipulates points, lines, shapes. These exercises mostly use the material of section 9.

#### 46.1 Basic functions

**Exercise 46.1.** Write a function with (float or double) inputs  $x, y$  that returns the distance of point  $(x, y)$  to the origin.

Test the following pairs: 1, 0; 0, 1; 1, 1; 3, 4.

**Exercise 46.2.** Write a function with inputs  $x, y, \theta$  that alters  $x$  and  $y$  corresponding to rotating the point  $(x, y)$  over an angle  $\theta$ .

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Your code should behave like:

**Code:**

```
// geom/rotate.cpp
const float pi = 2*acos(0.0);
float x{1.}, y{0.};
rotate(x,y,pi/4);
cout << "Rotated halfway: (" 
    << x << "," << y << ")" << '\n';
rotate(x,y,pi/4);
cout << "Rotated to the y-axis: (" 
    << x << "," << y << ")" << '\n';
```

**Output:**

```
[geom] rotate:
Rotated halfway:
    ↪(0.707107,0.707107)
Rotated to the y-axis: (0,1)
```

#### 46.2 Point class

*Before doing this section, make sure you study section 9.1.*

## 46. Geometry

---

A class can contain elementary data. In this section you will make a *Point* class that models Cartesian coordinates and functions defined on coordinates.

**Exercise 46.3.** Make class *Point* with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

Write the following methods:

- *distance\_to\_origin* returns a *float*.
- *angle* computes the angle of vector  $(x, y)$  with the *x*-axis.

**Exercise 46.4.** Extend the *Point* class of the previous exercise with a method: *distance* that computes the distance between this point and another: if  $p, q$  are *Point* objects,

```
p.distance(q)
```

computes the distance between them.

Hint: distance  $\Delta = \sqrt{\delta_x^2 + \delta_y^2}$ . Don't be afraid to introduce more methods than just *distance*.

**Exercise 46.5.** Write a method *translated* that, given a *Point* and two *floats*, returns the point translated by those amounts:

**Code:**

```
// geom/halfway.cpp
Point p1( 1.5, 2.5 );
cout << p1.stringifyed() << '\n';
float x=.2, y=.3;
auto p2 = p1.translated(x,y);
cout << "by: " << x << "," << y << '\n';
cout << p2.stringifyed() << '\n';
```

**Output:**

```
[geom] translate:
(1.5, 2.5)
by: 0.2, 0.3
(1.7, 2.8)
```

**Exercise 46.6.** Write a method *halfway* that, given two *Point* objects  $p, q$ , construct the *Point* halfway, that is,  $(p + q)/2$ :

```
Point p(1,2.2), q(3.4,5.6);
Point h = p.halfway(q);
```

You can write this function directly, or you could write functions *Add* and *Scale* and combine these. (Later you will learn about operator overloading.)

How would you print out a *Point* to make sure you compute the halfway point correctly?

**Exercise 46.7.** Make a default constructor for the point class:

```
Point() { /* default code */ }
```

which you can use as:

```
Point p;
```

but which gives an indication that it is undefined:

```
Code:
// geom/linear.cpp
Point p3;
cout << "Uninitialized point:"
     << '\n';
p3.printout();
cout << "Using uninitialized point:"
     << '\n';
auto p4 = Point(4,5)+p3;
p4.printout();
```

```
Output:
[geom] linearnan:
Uninitialized point:
Point: nan,nan
Using uninitialized point:
Point: nan,nan
```

Hint: see section 26.3.4.

**Exercise 46.8.** Revisit exercise 46.2 using the *Point* class. Your code should now look like:

```
newpoint = point.rotate(alpha);
```

**Exercise 46.9.** Advanced. Can you make a *Point* class that can accommodate any number of space dimensions? Hint: use a *vector*; section 10.3. Can you make a constructor where you do not specify the space dimension explicitly?

### 46.3 Using one class in another

Before doing this section, make sure you study section 9.2.

**Exercise 46.10.** Make a class *LinearFunction* with a constructor:

```
LinearFunction( Point input_p1, Point input_p2 );
```

and a member function

```
float evaluate_at( float x );
```

which you can use as:

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

**Exercise 46.11.** Make a class *LinearFunction* with constructor:

```
LinearFunction( Point input_p1, Point input_p2 );
```

where the first stands for a line through the origin.

Implement again the *evaluate* function so that

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

**Exercise 46.12.** Can you extend the previous exercise to let

```
LinearFunction line( p1 )
```

mean a line through the origin?

**Exercise 46.13.** Revisit exercises 46.2 and 46.8, introducing a `Matrix` class. Your code can now look like

```
newpoint = point.apply(rotation_matrix);
```

or

```
newpoint = rotation_matrix.apply(point);
```

Can you argue in favor of either one?

Suppose you want to write a `Rectangle` class, which could have methods such as `float Rectangle::area()` or `bool Rectangle::contains(Point)`. Since rectangle has four corners, you could store four `Point` objects in each `Rectangle` object. However, there is redundancy there: you only need three points to infer the fourth. Let's consider the case of a rectangle with sides that are horizontal and vertical; now you need only two points.

To implement a rectangle with sides parallel to the x/y axes, two designs are possible. For the function:

```
float Rectangle::area();
```

it is most convenient to store width and height.

For inclusion testing:

```
bool Rectangle::contains(Point);
```

it would be convenient to store bottomleft/topleft points.

For now, use the first option.

**Exercise 46.14.** Make a class `Rectangle` (sides parallel to axes) with a constructor:

```
Rectangle(Point botleft, float width, float height);
```

Can you figure out how to use *member initializer* lists for the constructors?

Implement methods:

```
float area(); float rightedge_x(); float topedge_y();
```

and write a main program to test these.

**Exercise 46.15.** Add a second constructor

```
Rectangle(Point botleft, Point topright);
```

Can you find a way to combine the constructors through *constructor delegating*? This can be done two

ways!

**Exercise 46.16.** Make a copy of your solution of the previous exercise, and redesign your class so that it stores two *Point* objects. Your main program should not change.

The previous exercise illustrates an important point: for well designed classes you can change the implementation (for instance motivated by efficiency) while the program that uses the class does not change.

## 46.4 Is-a relationship

*Before doing this section, make sure you study section 9.3.*

**Exercise 46.17.** Take your code where a *Rectangle* was defined from one point, width, and height.

Make a class *Square* that inherits from *Rectangle*. It should have the function *area* defined, inherited from *Rectangle*.

First ask yourself: what should the constructor of a *Square* look like?

**Exercise 46.18.** Revisit the *LinearFunction* class. Add methods *slope* and *intercept*.

Now generalize *LinearFunction* to *StraightLine* class. These two are almost the same except for vertical lines. The *slope* and *intercept* do not apply to vertical lines, so design *StraightLine* so that it stores the defining points internally. Let *LinearFunction* inherit.

## 46.5 Pointers

*Before doing this section, make sure you study section 16.1.*

**Exercise 46.19.** With this code given:

```
Code:
// pointer/dynrectangle.cpp
float dx( Point other ) {
    return other.x-x; }
/* ... */
// main, with objects
Point
    oneone(1,1), fivetwo(5,2);
float dx = oneone.dx(fivetwo);
/* ... */
// main, with pointers
auto
    oneonep = make_shared<Point>(1,1),
    fivetwop = make_shared<Point>(5,2);
```

```
Output:
[pointer] dynpoint:
dx: 4
dx: 4
```

compute the *dx* between the *oneonep* & *fivetwop*.

You can base this off the file *dynrectangle.cxx* in the repository

**Exercise 46.20.** Make a `DynRectangle` class, which is constructed from two shared-pointers-to-`Point` objects:

```
// pointer/dynrectangle.cpp
auto
origin  = make_shared<Point>(0,0),
fivetwo = make_shared<Point>(5,2);
DynRectangle lielow( origin,fivetwo );
```

**Exercise 46.21.** Test this design: Calculate the area, scale the top-right point, and recalculate the area:

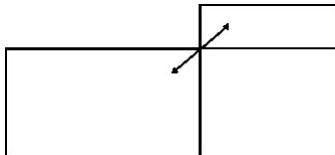
**Code:**

```
// pointer/dynrectangle.cpp
cout << "Area: " << lielow.area() << '\n';
/* ... */
cout << "Area: " << lielow.area() << '\n';
```

**Output:**

[pointer] dynrect:

```
Area: 10
Area: 40
```



**Exercise 46.22.** Make two `DynRectangle` objects so that the top-right corner of the first is the bottom-left corner of the other.

Now shift that point. Print out the two areas before and after to check correct behavior.

## 46.6 Operator overloading

The `Point` class has methods for adding and scaling points:

```
Point p3 = p1.add(p2);
Point p4 = p3.scale(2.5);
```

Wouldn't it be cool if you could write

```
Point p3 = p1+p2;
Point p4 = p3*2.5;
```

This is possible because you can *overload* the operators. For instance,

```
// geom/overload.cpp
Point operator*( float f ) {
    return Point( x*f, y*f );
}
```

**Exercise 46.23.** Define the plus operator between `Point` objects. The declaration is:

```
Point operator+(Point q);
```

You can base this off the file `overload.cxx` in the repository

**Exercise 46.24.** Rewrite the `halfway` method of exercise 46.6 and replace the `add` and `scale` functions by overloaded operators.

Hint: for the `add` function you may need ‘`this`’.

**Exercise 46.25.**

Evaluate a linear function:

Using method:

```
// geom/overload.cpp
LinearFunction line(p1, p2);
cout << "Value at 4.0: "
    << line.evaluate_at(4.0)
    << '\n';
```

using operator:

```
// geom/overload.cpp
float y = line(4.0);
cout << y << '\n';
```

Write the appropriate overloaded operator.

You can base this off the file `overload.cxx` in the repository

## 46.7 More stuff

Before doing this section, make sure you study section 15.3.

The `Rectangle` class stores at most one corner, but may be convenient to sometimes have an array of all four corners.

**Exercise 46.26.** Add a method

```
const vector<Point> &corners()
```

to the `Rectangle` class. The result is an array of all four corners, not in any order. Show by a compiler error that the array can not be altered.



## Chapter 47

### Zero finding

#### 47.1 Root finding by bisection

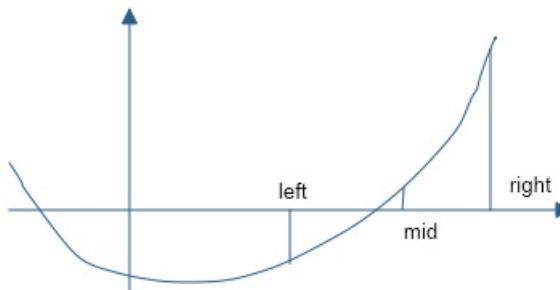


Figure 47.1: Root finding by interval bisection

For many functions  $f$ , finding their zeros, that is, the values  $x$  for which  $f(x) = 0$ , can not be done analytically. You then have to resort to numerical *root finding* schemes. In this project you will develop gradually more complicated implementations of a simple scheme: root finding by *bisection*.

In this scheme, you start with two points where the function has opposite signs, and move either the left or right point to the mid point, depending on what sign the function has there. See figure 47.1.

**Remark** In section 47.2 we will then look at Newton's method. This method is mathematically superior, so the current discussion of bisection should only be considered as a programming exercise.

##### 47.1.1 Simple implementation

Before doing this section, make sure you study section 7.

Before doing this section, make sure you study section 10.

Let's develop a first implementation step by step. To ensure correctness of our code we will use a Test-Driven Development (TDD) approach: for each bit of functionality we write a test to ensure its correctness before we integrate it in the larger code. (For more about TDD, and in particular the Catch2 framework, see section 68.2.)

### 47.1.2 Polynomials

First of all, we need to have a way to represent polynomials. For a polynomial of degree  $d$  we need  $d + 1$  coefficients:

$$f(x) = c_0x^d + \cdots + c_{d-1}x^1 + c_d \quad (47.1)$$

We implement this by storing the coefficients in a `vector<double>`. We make the following arbitrary decisions

1. let the first element of this vector be the coefficient of the highest power, and
2. for the coefficients to properly define a polynomial, this leading coefficient has to be nonzero.

Let's start by having a fixed test polynomial, provided by a function `set_coefficients`. For this function to provide a proper polynomial, it has to satisfy the following test:

```
// root/testzeroarray.cpp
TEST_CASE( "coefficients represent polynomial" "[1]" ) {
    vector<double> coefficients = { 1.5, 0., -3 };
    REQUIRE( coefficients.size() > 0 );
    REQUIRE( coefficients.front() != 0. );
}
```

**Exercise 47.1.** Write a routine `set_coefficients` that constructs a vector of coefficients:

```
// root/findzeroarray.cpp
vector<double> coefficients = set_coefficients();
```

and make it satisfy the above conditions.

At first write a hard-coded set of coefficients, then try reading them from the command line.

**Exercise 47.2.** Bonus: use the `cxxopts` library (section 63.2.2) to specify the coefficients on the commandline.

Above we postulated two conditions that an array of numbers should satisfy to qualify as the coefficients of a polynomial. Your code will probably be testing for this, so let's introduce a boolean function `is_proper_polynomial`:

- This function returns `true` if the array of numbers satisfies the two conditions;
- it returns `false` if either condition is not satisfied.

In order to test your function `is_proper_polynomial` you should check that

- it recognizes correct polynomials, and
- it fails for improper coefficients that do not properly define a polynomial.

**Exercise 47.3.** For polynomial coefficients to give a well-defined polynomial, the zero-th coefficient needs to be non-zero:

```
// bisect/zeroclasstest.cpp
TEST_CASE( "proper test", "[2]" ) {
    vector<double> coefficients{3., 2.5, 2.1};
    REQUIRE_NO_THROW( polynomial(coefficients) );

    coefficients.at(0) = 0.;
    REQUIRE_THROWS( polynomial(coefficients) );
```

```
}
```

Write a constructor that accepts the coefficients, and throws an exception if the above condition is violated.

We need polynomial evaluation. We will build a function `evaluate_at` with the following definition:

```
// root/findzerolib.hpp
double evaluate_at( const std::vector<double>& coefficients, double x);
```

You can interpret the array of coefficients in (at least) two ways, but with equation (47.1) we proscribed one particular interpretation.

So we need a test that the coefficients are indeed interpreted with the leading coefficient first, and not with the leading coefficient last. For instance:

```
// root/testzeroclass.cpp
polynomial second( {2, 0, 1} );
// correct interpretation: 2x^2 + 1
REQUIRE( second.is_proper() );
REQUIRE( second.evaluate_at(2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( second.evaluate_at(2) != Catch::Approx(6) );
```

(where we have left out the `TEST_CASE` header.)

Now we write the function that passes these tests:

**Exercise 47.4.** Write a function `evaluate_at` which computes

$$y \leftarrow f(x).$$

and confirm that it passes the above tests.

```
double evaluate_at( polynomial coefficients, double x);
```

For bonus points, look up *Horner's rule* and implement it.

With the polynomial function implemented, we can start working towards the algorithm.

### 47.1.3 Left/right search points

Suppose  $x_-, x_+$  are such that

$$x_- < x_+, \quad \text{and} \quad f(x_-) \cdot f(x_+) < 0,$$

that is, the function values in the left and right point are of opposite sign. Then there is a zero in the interval  $(x_-, x_+)$ ; see figure 47.1.

But how to find these outer bounds on the search?

If the polynomial is of odd degree you can find  $x_-, x_+$  by going far enough to the left and right from any two starting points. For even degree there is no such simple algorithm (indeed, there may not be a zero) so we abandon the attempt.

We start by writing a function `is_odd` that tests whether the polynomial is of odd degree.

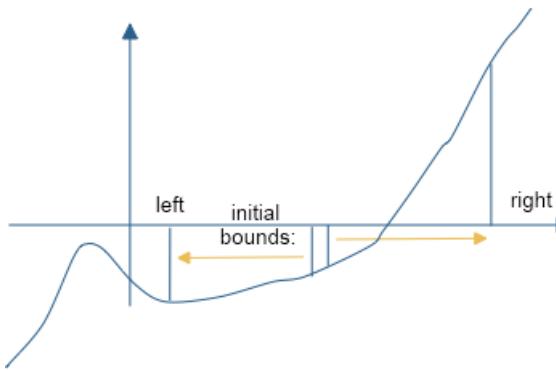


Figure 47.2: Setting the initial search points

**Exercise 47.5.** Write a method `is_odd` that makes the following test pass:

```
// bisect/zeroclasstest.cpp
TEST_CASE( "polynomial degree", "[3]" ) {
    polynomial second( {2,0,1} ); // 2x^2 + 1
    REQUIRE( not second.is_odd() );
    polynomial third( {3,2,0,1} ); // 3x^3 + 2x^2 + 1
    REQUIRE( third.is_odd() );
}
```

Now we can find  $x_-$ ,  $x_+$ : start with some interval and move the end points out until the function values have opposite sign.

**Exercise 47.6.** Write a function `find_initial_bounds` which computes  $x_-$ ,  $x_+$  such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

Address the following concerns:

1. What is a good prototype for the function?
2. How do you move the points far enough out to satisfy this condition?
3. Can you compute the above test more compactly?

Since finding a left and right point with a zero in between is not always possible for polynomials of even degree, we completely reject this case. In the following test we throw an exception (see section 23.2, in particular 23.2.3) for polynomials of even degree:

```
// root/testzeroarray.cpp
right = left+1;
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_initial_bounds(second, left, right) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_initial_bounds(third, left, right) );
REQUIRE( left<right );
```

Make sure your code passes these tests. What test do you need to add for the function values?

#### 47.1.4 Root finding

The root finding process globally looks as follows:

- You start with points  $x_-, x_+$  where the function has opposite sign; then you know that there is a zero between them.
- The bisection method for finding this zero looks at the halfway point, and based on the function value in the mid point:
- moves one of the bounds to the mid point, such that the function again has opposite signs in the left and right search point.

The structure of the code is as follows:

```
double find_zero( /* something */ ) {
    while ( /* left and right too far apart */ ) {
        // move bounds left and right closer together
    }
    return something;
}
```

Again, we test all the functionality separately. In this case this means that moving the bounds should be a testable step.

**Exercise 47.7.** Write a function `move_bounds_closer` and test it.

```
void move_bounds_closer
( std::vector<double> coefficients,
  double& left, double& right );
```

Implement some unit tests on this function.

Finally, we put everything together in the top level function `find_zero`.

**Exercise 47.8.** Make this call work:

```
// root/findzeroarray.cpp
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients, zero) << '\n';
```

Design unit tests, including on the precision attained, and make sure your code passes them.

#### 47.1.5 Object implementation

Revisit the exercises of section 47.1.1 and introduce a `polynomial` class that stores the polynomial coefficients. Several functions now become members of this class.

Also update the unit tests.

Some further suggestions:

1. Can you make your polynomial class look like a function?

```
class Polynomial {
    /* ... */
}
```

```
main () {
    Polynomial p;
    float y = p(x);
}
```

See section 9.5.7.1.

2. Can you generalize the polynomial class, for instance to the case of special forms such as  $(1+x)^n$ ?
3. Templatize your polynomials: Make a templated version that works both with `float` and `double`. Can you see a difference in attainable precision between the two types?

## 47.2 Newton's method

*Before doing this section, make sure you study section 13.*

In this section we look at *Newton's method*. This is an iterative method for finding zeros of a function  $f$ , that is, it computes a sequence of values  $\{x_n\}$ , so that  $f(x_n) \rightarrow 0$ . The sequence is defined by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

with  $x_0$  arbitrarily chosen. For details, see HPC book [13], appendix 23.

While in practice Newton's method is used for complicated functions, here we will look at a simple example, which actually has a basis in computing history. Early computers had no hardware for computing a square root. Instead, they used Newton's method.

Suppose you have a positive value  $y$  and you want want to compute  $x = \sqrt{y}$ . This is equivalent to finding the zero of

$$f(x) = x^2 - y$$

where  $y$  is fixed. To indicate this dependence on  $y$ , we will write  $f_y(x)$ . Newton's method then finds the zero by evaluating

$$x_{\text{next}} = x - f_y(x)/f'_y(x)$$

until the guess is accurate enough, that is, until  $f_y(x) \approx 0$ .

We will not go into the matter of how to choose a sophisticated stopping test for the iteration; that is a matter for a numerical analysis course, not a programming course.

### 47.2.1 Function implementation

We start with the special case of  $\sqrt{2}$ , that is, the function  $f(x) = x^2 - 2$ .

It is of course simple to code a Newton's method for such a special case: it should take you about 10 lines. Ultimately we want to have a general code that takes any two functions  $f, f'$ , and then uses Newton's method to find a zero of our specific function  $f$ . We get there in steps.

**Exercise 47.9.** Compute  $\sqrt{2}$  as the zero of  $f_y(x) = x^2 - y$  for the special case of  $y = 2$ .

- Write functions `f(x)` and `deriv(x)`, that compute  $f_y(x)$  and  $f'_y(x)$  for the particular definition of  $f_y$ .
- Iterate until  $|f(x, y)| < 10^{-5}$ . Print  $x$  and  $f(x)$  in each iteration; don't worry too much about the stopping test and accuracy attained.
- Second part: write a function `newton_root` that computes  $\sqrt{y}$  again: only for  $\sqrt{2}$ .

### 47.2.2 Using lambda expressions

Above you wrote functions with a declaration:

```
// newton/newton-fun.cpp
double f(double x);
double fprime(double x);
```

and the algorithm:

```
// newton/newton-fun.cpp
double x{1.};
while (true) {
    auto fx = f(x);
    cout << "f( " << x << " ) = " << fx << '\n';
    if (std::abs(fx)<1.e-10) break;
    x = x - fx/fprime(x);
}
```

We will now start introducing lambda expression, rather than using named functions. For more on lambda expressions, see chapter 13.

**Exercise 47.10.** Rewrite your code to use lambda functions for `f` and `fprime`.

If you use variables for the lambda expressions, put them in the main program.

You can base this off the file `newton.cxx` in the repository

Next, we make the code modular by writing a general function `newton_root`, that contains the Newton method of the previous exercise. Since it has to work for any functions  $f, f'$ , you have to pass the objective function and the derivative as arguments:

```
double root = newton_root( f, fprime );
```

**Exercise 47.11.** Rewrite the Newton exercise by implementing a `newton_root` function:

```
double root = newton_root( f, fprime );
```

Call the function

1. first with the lambda variables you already created;
2. then directly with the lambda expressions as arguments, that is, without assigning them to variables.

Next we extend the functionality of our code to finding the square root of any nonnegative number, not by changing the objective function  $f$ . Instead, we use a more general way of specifying the objective function and derivative.

**Exercise 47.12.** Extend the Newton exercise to compute roots in a loop:

```
// newton/newton-lambda.cpp
for ( float n=2.; n<10; n:=.5 ) {
    cout << "sqrt(" << n << ") = "
    << newton_root(
/* ... */
)
    << '\n';
}
```

Without lambdas, you would define a function

```
double squared_minus_n( double x,int n ) {
    return x*x-n;
}
```

However, the `newton_root` function takes a function of only a real argument. Use a capture to make  $f$  dependent on the integer parameter.

It is possible to dispense with the derivative function by using numerical differentiation: we use the limit formula for the derivative, and use a finite difference, rather than a limit. The result is an ‘inexact Newton’s method’, and mathematical theory says that this will also converge if the inexactness is not too bad, though it may converge slower. This is a topic for numerical analysis [19, 14] which we will not go into here. We will use this fact to simplify our Newton code.

You can approximate the derivative of a function  $f$  as

$$f'(x) = (f(x + h) - f(x))/h$$

where  $h$  is small.

This is called a ‘finite difference’ approximation.

This looks like the limit equation for computing the derivative, but in code we can not take a limit, so by taking a finite  $h$  we are computing a ‘numerical derivative’ here. See a numerical analysis textbook for discussion.

**Exercise 47.13.** Write a version of the root finding function that only takes the objective function:

```
double newton_root( function< double(double)> f )
```

and approximates the derivative by a finite difference. You can use a fixed value  $h=1e-6$ .

Do not reimplement the whole newton method: instead create a lambda expression for the gradient and pass it to the function `newton_root` you coded earlier as second argument.

**Exercise 47.14.** Bonus: can you compute logarithms through Newton’s method?

### 47.2.3 Templatized implementation

Newton's method works equally well for complex numbers as for real numbers.

**Exercise 47.15.** Rewrite your Newton code where the algorithm is in the main program so that it works for complex numbers. Here is the main; you need to write the functions:

```
// newton/newton-complex.cpp


You may run into the problem that you can not operate immediately between a complex number and a float or double. Use static_cast:


```

```
static_cast< complex<double> >(2)
```

So do you have to write two separate implementations, one for reals, and one for complex numbers? (And maybe you need two separate ones for `float` and `double`!) This is where *templates* come in handy.

You can templatize the objective function and the derivative:

```
// newton/newton-double.cpp
template<typename T>
T f(T x) { return x*x - 2; }
template<typename T>
T fprime(T x) { return 2 * x; }
```

and then write in the main program:

```
// newton/newton-double.cpp
double x{1.};
while ( true ) {
    auto fx = f<double>(x);
    cout << "f( " << x << " ) = " << fx << '\n';
    if (std::abs(fx)<1.e-10) break;
    x = x - fx/fprime<double>(x);
}
```

**Exercise 47.16.** Update your `newton_root` function with a template parameter. Test it by having a main program that computes some roots in `float`, `double`, and `complex<double>`.

**Exercise 47.17.** Use your complex Newton method to compute  $\sqrt{2}$ . Does it work?

How about  $\sqrt{-2}$ ? If it doesn't work, why, and can you fix this?

There is no reason to limit the template parameter to the type of the function argument. You could also use a template parameter for the functions:

```
template< typename F >
void call_on_5( F f ) {
    f(5);
}
```

Now apply this idea to the Newton function.

**Exercise 47.18.** Write a Newton method where the objective function is itself a template parameter, not just its arguments and return type. Hint: no changes to the main program are needed.

Then compute  $\sqrt{-2}$  as:

```
// newton/lambda-complex.cpp
cout << "sqrt -2 = " <<
newton_root<complex<double>>
( // objective function
[] (complex<double> x) -> complex<double> {
    return x*x + static_cast<complex<double>>(2); },
// derivative
[] (complex<double> x) -> complex<double> {
    return x * static_cast<complex<double>>(2); },
// initial value
complex<double>{.1,.1}
)
<< '\n';
```

This may look strange: how do you enforce that the functions are of a type that fits with the scalar type? For instance, you may write mismatch the routine type and the types of the functions:

```
// newton/newton-templatetf.cpp
complex<double> sgrte2 =
newton_root<double>
( // objective function
[two] ( complex<double> x ) -> complex<double> {
    return x*x - static_cast<complex<double>>(-two); },
// derivative
[] ( complex<double> x ) -> complex<double> {
    return static_cast<complex<double>>(2) * x; },
// initial value
{1.,1.}
);
```

And the answer is that you don't have to enforce this: you leave it up to the compiler to complain if there is a mismatch.

In the previous exercise you were told to use main program unchanged, meaning that you still call the newton routine as:

```
newton_root<complex<double>>
```

Did you wonder that you specify only one template parameter here, while the routine is defined with three? The compiler is very good at ‘argument dependent lookup’, where it figures out the missing template parameters from the one you specify and the code.

## Chapter 48

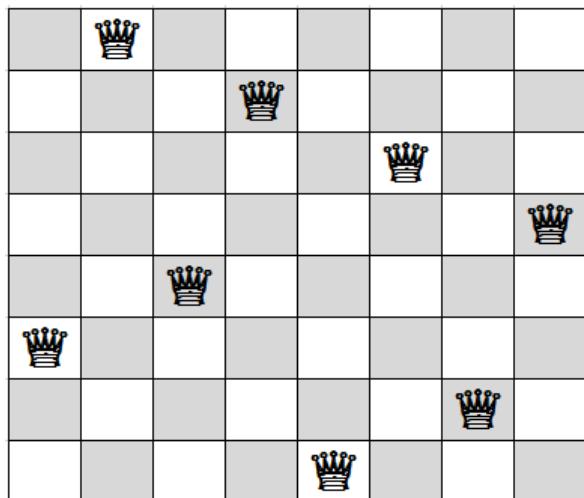
### Eight queens

A famous exercise for recursive programming is the *eight queens* problem: Is it possible to position eight queens on a chess board, so that no two queens ‘threaten’ each other, according to the rules of chess?

#### 48.1 Problem statement

The precise statement of the ‘eight queens problem’ is:

- Put eight pieces on an  $8 \times 8$  board, no two pieces on the same square; so that
- no two pieces are on the same row,
- no two pieces are on the same column, and
- no two pieces are on the same diagonal.



A systematic solution would run:

1. put a piece anywhere in the first row;
2. for each choice in the first row, try all positions in the second row;
3. for all choices in the first two rows, try all positions in the third row;
4. when you have a piece in all eight rows, evaluate the board to see if it satisfies the condition.

**Exercise 48.1.** This algorithm will generate all  $8^8$  boards. Do you see at least one way to speed up the search?

Since the number eight is, for now, fixed, you could write this code as an eight-deep loop nest. However that is not elegant. For example, the only reason for the number 8 in the above exposition is that this is the traditional size of a chess board. The problem, stated more abstractly as placing  $n$  queens on an  $n \times n$  board, has solutions for  $n \geq 4$ .

## 48.2 Solving the eight queens problem, basic approach

This problem requires you to know about arrays/vectors; chapter 10. Also, see chapter 68 for TDD. Finally, see section 24.6.2 for `std::optional`.

The basic strategy will be that we fill consecutive rows, by indicating each time which column will be occupied by the next queen. Using an Object-Oriented (OO) strategy, we make a class `ChessBoard`, that holds a partially filled board.

The basic solution strategy is recursive:

- Let `current` be the current, partially filled board;
- We call `current.place_queens()` that tries to finish the board;
- However, with recursion, this method only fills one row, and then calls `place_queens` on this new board.

So

```
ChessBoard::place_queens() {
    // for c = 1 ... number of columns:
    //  make a copy of the board
    //  put a queen in the next row, column c, of the copy
    //  and call place_queens() on that copy;
    //  investigate the result.....
}
```

This routine returns either a solution, or an indication that no solution was possible.

In the next section we will develop a solution systematically in a TDD manner.

## 48.3 Developing a solution by TDD

We now gradually develop the OO solution to the eight queens problem, using test-driven development.

**The board** We start by constructing a board, with a constructor that only indicates the size of the problem:

```
// queens/queens.hpp
ChessBoard(int n);
```

This is a ‘generalized chess board’ of size  $n \times n$ , and initially it should be empty.

**Exercise 48.2.** Write this constructor, for an empty board of size  $n \times n$ .

Note that the implementation of the board is totally up to you. In the following you will get tests for functionality that you need to satisfy, but any implementation that makes this true is a correct solution.

**Bookkeeping: what's the next row?** Assuming that we fill in the board row-by-row, we have an auxiliary function that returns the next row to be filled:

```
// queens/queens.hpp
int next_row_to_be_filled()
```

This gives us our first simple test: on an empty board, the row to be filled in is row zero.

**Exercise 48.3.** Write this method and make sure that it passes the test for an empty board.

```
// queens/queentest.cpp
TEST_CASE( "empty board", "[1]" ) {
    constexpr int n=10;
    ChessBoard empty(n);
    REQUIRE( empty.next_row_to_be_filled() == 0 );
}
```

By the rules of TDD you can actually write the method so that it only satisfies the test for the empty board. Later, we will test that this method gives the right result after we have filled in a couple of rows, and then of course your implementation needs to be general.

**Place one queen** Next, we have a function to place the next queen, whether this gives a feasible board (meaning that no pieces can capture each other) or not:

```
// queens/queens.hpp
void place_next_queen_at_column(int i);
```

This method should first of all catch incorrect indexing: we assume that the placement routine throws an exception for invalid column numbers.

```
ChessBoard::place_next_queen_at_column( int c ) {
    if ( /* c is outside the board */ )
        throw(1); // or some other exception.
```

(Suppose you didn't test for incorrect indexing. Can you construct a simple 'cheating' solution at any size?)

**Exercise 48.4.** Write this method, and make sure it passes the following test for valid and invalid column numbers:

```
// queens/queentest.cpp
INFO( "Illegal placement throws" )
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
INFO( "Correct placement succeeds" );
REQUIRE_NO_THROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled() == 1 );
```

(From now on we'll only give the body of the test.)

Now it's time to start writing some serious stuff.

**Is a (partial) board feasible?** If you have a board, even partial, you want to test if it's feasible, meaning that the queens that have been placed can not capture each other.

The prototype of this method is:

```
// queens/queens.hpp  
bool feasible()
```

This test has to work for simple cases to begin with: an empty board is feasible, as is a board with only one piece.

```
// queens/queentest.cpp  
ChessBoard empty(n);  
REQUIRE( empty.feasible() );  
  
// queens/queentest.cpp  
ChessBoard one = empty;  
one.place_next_queen_at_column(0);  
REQUIRE( one.next_row_to_be_filled()==1 );  
REQUIRE( one.feasible() );
```

**Exercise 48.5.** Write the method and make sure it passes these tests.

We shouldn't only do successful tests, sometimes referred to as the 'happy path' through the code. For instance, if we put two queens in the same column, the test should fail.

**Exercise 48.6.** Take the above initial attempt with a queen in position (0, 0), and add another queen in column zero of the next row. Check that it passes the test:

```
// queens/queentest.cpp  
ChessBoard collide = one;  
// place a queen in a 'colliding' location  
collide.place_next_queen_at_column(0);  
// and test that this is not feasible  
REQUIRE( not collide.feasible() );
```

Add a few tests more of your own. (These will not be exercised by the submission script, but you may find them useful anyway.)

**Testing configurations** If we want to test the feasibility of non-trivial configurations, it is a good idea to be able to 'create' solutions. For this we need a second type of constructor where we construct a fully filled chess board from the locations of the pieces.

```
// queens/queens.hpp  
ChessBoard( int n, vector<int> cols );  
ChessBoard( vector<int> cols );
```

- If the constructor is called with only a vector, this describes a full board.

- Adding an integer parameter indicates the size of the board, and the vector describes only the rows that have been filled in.

**Exercise 48.7.** Write these constructors, and test that an explicitly given solution is a feasible board:

```
// queens/queentest.cpp
ChessBoard five( {0, 3, 1, 4, 2} );
REQUIRE( five.feasible() );
```

For an elegant approach to implementing this, see *delegating constructors*; section 9.4.1.

Ultimately we have to write the tricky stuff.

## 48.4 The recursive solution method

The main function

```
// queens/queens.hpp
optional<ChessBoard> place_queens()
```

takes a board, empty or not, and tries to fill the remaining rows.

One problem is that this method needs to be able to communicate that, given some initial configuration, no solution is possible. For this, we let the return type of `place_queens` be `optional<ChessBoard>`:

- if it is possible to finish the current board resulting in a solution, we return that filled board;
- otherwise we return `{}`, indicating that no solution was possible.

With the recursive strategy discussed in section 48.2, this placement method has roughly the following structure:

```
place_queens() {
    for ( int col=0; col<n; col++ ) {
        ChessBoard next = *this;
        // put a queen in column col on the 'next' board
        // if this is feasible and full, we have a solution
        // if it is feasible but no full, recurse
    }
}
```

The line

```
ChessBoard next = *this;
```

makes a copy of the object you're in.

**Remark** Another approach would be to make a recursive function

```
bool place_queen( const ChessBoard& current, ChessBoard &next );
// true if possible, false is not
```

**The final step** Above you coded the method `feasible` that tested whether a board is still a candidate for a solution. Since this routine works for any partially filled board, you also need a method to test if you're done.

**Exercise 48.8.** Write a method

```
bool filled();
```

and write a test for it, both positive and negative.

Now that you can recognize solutions, it's time to write the solution routine.

**Exercise 48.9.** Write the method

```
// queens/queens.hpp
optional<ChessBoard> place_queens()
```

Because the function `place_queens` is recursive, it is a little hard to test in its entirety.

We start with a simpler test: if you almost have the solution, it can do the last step.

**Exercise 48.10.** Use the constructor

```
ChessBoard( int n, vector<int> cols )
```

to generate a board that has all but the last row filled in, and that is still feasible. Test that you can find the solution:

```
// queens/queentest.cpp
ChessBoard almost( 4, {1,3,0} );
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Since this test only fills in the last row, it only does one loop, so printing out diagnostics is possible, without getting overwhelmed in tons of output.

**Solutions and non-solutions** Now that you have the solution routine, test that it works starting from an empty board. For instance, confirm there are no  $3 \times 3$  solutions:

```
// queens/queentest.cpp
TEST_CASE( "no 3x3 solutions", "[9]" ) {
    ChessBoard three(3);
    auto solution = three.place_queens();
    REQUIRE( not solution.has_value() );
}
```

On the other hand,  $4 \times 4$  solutions do exist:

```
// queens/queentest.cpp
TEST_CASE( "there are 4x4 solutions", "[10]" ) {
    ChessBoard four(4);
    auto solution = four.place_queens();
    REQUIRE( solution.has_value() );
}
```

**Exercise 48.11.** (Optional) Can you modify your code so that it counts all the possible solutions?

**Exercise 48.12.** (Optional) How does the time to solution behave as function of  $n$ ?



## Chapter 49

### Infectious disease simulation

In recent years, the spread of infectious diseases has been a hot topic. First there was the anti-vaccination movement, and then people had to deal with the outbreak of the Corona virus for which initially no vaccine even existed. Apart from any sociological angle, this issue can be tackled by any number of sciences. In this project we will model how fast a disease spreads through a partly vaccinated population, depending on various parameters. The approach taken here is one of explicitly modeling a population through its members, rather than taking a statistical approach.

This section contains a sequence of exercises that builds up to a somewhat realistic simulation of infectious disease propagation.

#### 49.1 Model design

It is possible to model disease propagation statistically, but here we will build an explicit simulation: we will maintain an explicit description of all the people in the population, and track for each of them their status.

We will use a simple model where a person can be:

- sick: when they are sick, they can infect other people;
- susceptible: they are healthy, but can be infected;
- recovered: they have been sick, but no longer carry the disease, and can not be infected for a second time;
- vaccinated: they are healthy, do not carry the disease, and can not be infected.

In more complicated models a person could be infectious during only part of their illness, or there could be secondary infections with other diseases, et cetera. We keep it simple here: any sick person can infect others while they are sick.

In the exercises below we will gradually develop a model of how the disease spreads from an initial source of infection. The program will then track the population from day to day, running indefinitely until none of the population is sick. Since there is no re-infection, the run will always end. Later we will add mutation to the model which can extend the duration of the epidemic.

### 49.1.1 Other ways of modeling

Instead of capturing every single person in code, a ‘contact network’ model, it is possible to use an ODE approach to disease modeling. You would then model the percentage of infected persons by a single scalar, and derive relations for that and other scalars [2, 20].

This is known as a ‘compartmental model’, where each of the three SIR states, Susceptible, Infected, Removed, is a compartment: a section of the population. Both the contact network and the compartmental model capture part of the truth. In fact, they can be combined. We can consider a country as a set of cities, where people travel between any pair of cities. We then use a compartmental model inside a city, and a contact network between cities.

In this project we will only use the network model.

## 49.2 Coding

The following sections are a step-by-step development of the code for this project. Later we will discuss running this code as a scientific experiment.

**Remark** *In various places you need a random number generator. It is preferred that you use the C++ native one, rather than one from the C language.*

### 49.2.1 Person basics

The most basic component of a disease simulation is to infect a person with a disease, and see the time development of that infection. Thus you need a person class and a disease class.

Before you start coding, ask yourself what behaviors these classes need to support.

- Person. The basic methods for a person are
  1. Get infected;
  2. Get vaccinated; and
  3. Progress by one day.

Furthermore you may want to query what the state of the person is: are they healthy, sick, recovered?

- Disease. For now, a disease itself doesn’t do much. (Later in the project you may want it to have a method `mutate`.) However, you may want to query certain properties:
  1. Chance of transmission; and
  2. Number of days a person stays sick when infected.

A test for a single person could have output along the following lines:

```
On day 10, Joe is susceptible
On day 11, Joe is susceptible
On day 12, Joe is susceptible
On day 13, Joe is susceptible
On day 14, Joe is sick (5 days to go)
On day 15, Joe is sick (4 days to go)
On day 16, Joe is sick (3 days to go)
On day 17, Joe is sick (2 days to go)
On day 18, Joe is sick (1 days to go)
On day 19, Joe is recovered
```

**Exercise 49.1.** Write a *Person* class with methods:

- *status\_string()* : returns a description of the person's state as a *string*;
  - *one\_more\_day()* : update the person's status to the next day;
  - *infect(s)* : infect a person with a disease, where the disease object  
*Disease s(n);*
- is specified to run for *n* days.

Your main program could for instance look like:

```
// infect/person.cpp
for ( int step = 1; ; ++step ) {

    joe.one_more_day();
    /* ... */
    cout << "On day " << step << ", Joe is "
       << joe.status_string() << '\n';
    if (joe.is_recovered())
        break;
}
```

where the infection part has been left out.

Your main concern is how to model the internal state of a person. This is actually two separate issues:

1. the state, and
2. if sick, how many days to go to recover.

You can find a way to implement this with a single integer, but it's better to use two. Also, write enough support methods such as the *is\_recovered* test.

#### 49.2.1.1 Person tests

It is easy to write code that seems to do the right thing, but does not behave correctly in all cases. So it is a good idea to subject your code to some systematic tests.

Make sure your *Person* objects pass these tests:

- After being infected with a 100% transmittable disease, they should register as sick.
- If they are vaccinated or recovered, and they come in contact with such a disease, they stay in their original state.
- If a disease has a transmission chance of 50%, and a number of people come into contact with it, about half of them should get sick. This test maybe a little tricky to write.

Can you use the *Catch2* unit testing framework? See section 68.2.

#### 49.2.2 Interaction

Next we model interactions between people: one person is healthy, another is infected, and when the two come into contact the disease may be transferred.

```
// infect/interaction.cpp
Person infected, healthy;
infected.infect(flu);
/* ... */
healthy.touch(infected);
```

The disease has a certain probability of being transferred, so you need to specify that probability. You could let the declaration be:

```
Disease flu( 5, 0.3 );
```

where the first parameter is the number of days an infection lasts, and the second the transfer probability.

**Exercise 49.2.** Add a transmission probability to the *Disease* class, and add a *touch* method to the *Person* class. Design and run some tests.

**Exercise 49.3.** Bonus: can you get the following disease specification to work?

```
// infect/interaction.cpp
Disease flu;
flu.duration() = 20;
flu.transfer_probability() = p;
```

Why could you consider this better than the earlier suggested syntax?

### 49.2.2.1 Interaction tests

Adapt the above tests, but now a person comes in contact with an infected person, rather than directly with a disease.

### 49.2.3 Population

Next we need a *Population* class, where a population contains a *vector* consisting of *Person* objects. Initially we only infect one person, and there is no transmission of the disease.

The *Population* class should at least have the following methods:

- *random\_infection* to start out with an infected segment of the population;
- *random\_vaccination* to start out with a number of vaccinated individuals.
- counting functions *count\_infected* and *count\_vaccinated*.

To run a realistic simulation you also need a *one\_more\_day* method that takes the population through a day. This is the heart of your code, and we will develop this gradually in the next section.

### 49.2.3.1 Population tests

Most population testing will be done in the following section. For now, make sure you pass the following tests:

- With a vaccination percentage of 100%, everyone should indeed be vaccinated.

## 49.3 Epidemic simulation

To simulate the spread of a disease through the population, we need an update method that progresses the population through one day:

- Sick people come into contact with a number of other members of the populace;

- and everyone gets one day older, meaning mostly that sick people get one day closer to recovery.

We develop this in a couple of steps.

### 49.3.1 No contact

At first assume that people have no contact, so the disease ends with the people it starts with.

The trace output should look something like:

```
Size of
population?
In step 1 #sick: 1 : ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step 2 #sick: 1 : ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step 3 #sick: 1 : ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step 4 #sick: 1 : ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step 5 #sick: 1 : ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ?
In step 6 #sick: 0 : ? ? ? ? ? ? ? ? - ? ? ? ? ? ? ? ?
Disease ran its course by step 6
```

**Remark** Such a display is good for a sanity check on your program behavior. If you include such displays in your writeup, make sure to use a monospace font, and don't use a population size that needs line wrapping. In further testing, you should use large populations, but do not include these displays.

**Exercise 49.4.** Program a population without infection.

- Write the `Population` class. The constructor takes the number of people:

```
Population population(npeople);
```

- Write a method that infects a number of random people:

```
// infect/pandemic.cpp
population.random_infection(fever, initial_infect);
```

- Write a method `count_infected` that counts how many people are infected.
- Write an `one_more_day` method that updates all persons in the population.
- Loop the `one_more_day` method until no people are infected: the `Population::one_more_day` method should apply `Person::one_more_day` to all person in the population.

Write a routine that displays the state of the population, using for instance: ? for susceptible, + for infected, – for recovered.

#### 49.3.1.1 Tests

Test that for the duration of the disease, the number of infected people stays constant, and that the sum of healthy and infected people stays equal to the population size.

### 49.3.2 Contagion

This past exercise was too simplistic: the original patient zero was the only one who ever got sick. Now let's incorporate contagion, and investigate the spread of the disease from a single infected person.

We start with a very simple model of infection.

**Exercise 49.5.** Write a simulation where in each step the direct neighbors of an infected person can now get sick themselves.

Run a number of simulations with population sizes and contagion probabilities. Are there cases where people escape getting sick?

#### 49.3.2.1 Tests

Do some sanity tests:

- If one person is infected with a disease with  $p = 1$ , the next day there should be 3 people sick. Unless the infected person is the first or last: then there are two.
- If person 0 is infected, and  $p = 1$ , the simulation should run for a number of days equal to the size of the population.
- How is the previous case if  $p = 0.5$ ?

#### 49.3.3 Vaccination

**Exercise 49.6.** Incorporate vaccination: read another number representing the percentage of people that has been vaccinated. Choose those members of the population randomly.

Describe the effect of vaccinated people on the spread of the disease. Why is this model unrealistic?

#### 49.3.4 Spreading

To make the simulation more realistic, we let every sick person come into contact with a fixed number of random people every day. This gives us more or less the *SIR model* [26].

Set the number of people that a person comes into contact with, per day, to 6 or so. (You can also let this be an upper bound for a random value, but that does not essentially change the simulation.) You have already programmed the probability that a person who comes in contact with an infected person gets sick themselves. Again start the simulation with a single infected person.

**Exercise 49.7.** Code the random interactions. Now run a number of simulations varying

- The percentage of people vaccinated, and
- the chance the disease is transmitted on contact.

Record how long the disease runs through the population. With a fixed number of contacts and probability of transmission, how is this number of function of the percentage that is vaccinated?

Report this function as a table or graph. Make sure you have enough data points for a meaningful conclusion. Use a realistic population size. You can also do multiple runs and report the average, to even out the effect of the random number generator.

**Exercise 49.8.** Investigate the matter of ‘herd immunity’: if enough people are vaccinated, then some people who are not vaccinated will still never get sick. Let’s say you want to have the probability of being not vaccinated, yet never getting sick, to be over 95 percent. Investigate the percentage of

vaccination that is needed for this as a function of the contagiousness of the disease.

As in the previous exercise, make sure your data set is large enough.

**Remark** The screen output you used above is good for sanity checks on small problems. However, for realistic simulations you have to think what is a realistic population size. If your university campus is a population where random people are likely to meet each other, what would be a population size to model that? How about the city where you live?

Likewise, if you test different vaccination rates, what granularity do you use? With increases of 5 or 10 percent you can print all results to your screen, but you may miss things. Don't be afraid to generate large amount of data and feed them directly to a graphing program.

#### 49.3.4.1 Sanity tests

As your model gets more realistic it becomes harder to test it fully, if only because of the use of random numbers. Still ...

- With zero vaccination, the disease should run for some logarithm of the size of the population.
- With hundred percent vaccination, apart from one sick person, the disease will run for as long as that person is sick.

### 49.3.5 Mutation

The Covid years have shown how important mutations of an original virus can be. Next, you can include mutation in your project. We model this as follows:

- Every so many transmissions, a virus will mutate into a new variant.
- A person who has recovered from one variant is still susceptible to other variants.
- For simplicity assume that each variant leaves a person sick the same number of days, and
- Vaccination is all-or-nothing: one vaccine is enough to protect against all variant;
- On the other hand, having recovered from one variant is not protection against others.

Implementation-wise speaking, we model this as follows. First of all, we need a `Disease` class, so that we can infect a person with an explicit virus;

```
// infect/infect.lib.hpp
void touch( const Person&, long int ps=0 );
void infect( const Disease& );
```

A `Disease` object now carries the information such as the chance of transmission, or how long a person stays under the weather. Modeling mutation is a little tricky. You could do it as follows:

- There is a global `variants` counter for new virus variants, and a global `transmissions` counter.
- Every time a person infects another, the newly infected person gets a new `Disease` object, with the current variant, and the transmissions counter is updated.
- There is a parameter that determines after how many transmissions the disease mutates. If there is a mutation, the global `variants` counter is updated, and from that point on, every infection is with the new variant. (Note: this is not very realistic. You are free to come up with a better model.)

- A each `Person` object has a vector of variants that they are recovered from; recovery from one variant only makes them immune from that specific variant, not from others.

**Exercise 49.9.** Add mutation to your model. Experiment with the mutation rate: as the mutation rate increases, the disease should stay in the population longer. Does the relation with vaccination rate change that you observed before?

#### 49.3.5.1 Tests

Implement some sanity tests.

Suppose a person never recovers from a disease (can you think of examples?). The number of mutations should increase roughly exponentially (why isn't the rate actually exponential?); test this.

### 49.3.6 Diseases without vaccine: Ebola and Covid-19

*This section is optional, for bonus points*

The project so far applies to diseases for which a vaccine is available, such as MMR for measles, mumps and rubella. The analysis becomes different if no vaccine exists, such as is the case for *Ebola* and *Covid-19*, as of this writing.

Instead, you need to incorporate ‘social distancing’ into your code: people do not get in touch with random others anymore, but only those in a very limited social circle. Design a model distance function, and explore various settings.

The difference between Ebola and Covid-19 is how long an infection can go unnoticed: the *incubation period*. With Ebola, infection is almost immediately apparent, so such people are removed from the general population and treated in a hospital. For Covid-19, a person can be infected, and infect others, for a number of days before they are sequestered from the population.

Add this parameter to your simulation and explore the behavior of the disease as a function of it.

## 49.4 Ethics

The subject of infectious diseases and vaccination is full of ethical questions. The main one is *The chances of something happening to me are very small, so why shouldn't I bend the rules a little?*. This reasoning is most often applied to vaccination, where people for some reason or other refuse to get vaccinated.

Explore this question and others you may come up with: it is clear that everyone bending the rules will have disastrous consequences, but what if only a few people did this?

## 49.5 Bonus: parallelism

The main computation in your code is the update of the population. Argue that this is a fully parallel operation. Use OpenMP to make the loop indeed parallel, and test the speedup you get.

Can you also express this as a range algorithm? Again test the speedup you get. On high core counts you may need to figure out the equivalent of `OMP_PROC_BIND=true`. This depends on how your compiler has implemented the parallel range algorithsm.

## 49.6 Bonus: testing

Read [10]. Can you test the hypothesis of that article?

## 49.7 Bonus: mathematical analysis

The SIR model can also be modeled through coupled difference or differential equations.

1. The number  $S_i$  of susceptible people at time  $i$  decreases by a fraction

$$S_{i+1} = S_i(1 - \lambda_i dt)$$

where  $\lambda_i$  is the product of the number of infected people and a constant that reflects the number of meetings and the infectiousness of the disease. We write:

$$S_{i+1} = S_i(1 - \lambda I_i dt)$$

2. The number of infected people similarly increases by  $\lambda S_i I_i$ , but it also decreases by people recovering (or dying):

$$I_{i+1} = I_i(1 + \lambda S_i dt - \gamma dt).$$

3. Finally, the number of ‘removed’ people equals that last term:

$$R_{i+1} = R_i(1 + \gamma I_i).$$

**Exercise 49.10.** Code this scheme. What is the effect of varying  $dt$ ?

**Exercise 49.11.** For the disease to become an epidemic, the number of newly infected has to be larger than the number of recovered. That is,

$$\lambda S_i I_i - \gamma I_i > 0 \Leftrightarrow S_i > \gamma / \lambda.$$

Can you observe this in your simulations?

The parameter  $\gamma$  has a simple interpretation. Suppose that a person stays ill for  $\delta$  days before recovering. If  $I_t$  is relatively stable, that means every day the same number of people get infected as recover, and therefore a  $1/\delta$  fraction of people recover each day. Thus,  $\gamma$  is the reciprocal of the duration of the infection in a given person.

## 49.8 Project writeup and submission

### 49.8.1 Program files

In the course of this project you have written more than one main program, but some code is shared between the multiple programs. Organize your code with one file for each main program, and a single ‘library’ file with the class methods.

You can do this two ways:

1. You make a ‘library’ file, say `infect_lib.cc`, and your main programs each have a line

```
#include "infect_lib.cc"
```

This is not the best solution, but it is acceptable for now.

2. The better solution requires you to use *separate compilation* for building the program, and you need a *header* file. You would now have `infect_lib.cc` which is compiled separately, and `infect_lib.h` which is included both in the library file and the main program:

```
#include "infect_lib.h"
```

See section 19.2.2 for more information.

Submit all source files with instructions on how to build all the main programs. You can put these instructions in a file with a descriptive name such as `README` or `INSTALL`, or you can use a *makefile*.

#### 49.8.2 Writeup

In the writeup, describe the ‘experiments’ you have performed and the conclusions you draw from them. The exercises above give you a number of questions to address.

For each main program, include some sample output, but note that this is no substitute for writing out your conclusions in full sentences.

The exercises in section 49.3.4 ask you to explore the program behavior as a function of one or more parameters. Include a table to report on the behavior you found. You can use Matlab or Matplotlib in Python (or even Excell) to plot your data, but that is not required.

# Chapter 50

## Google PageRank

### 50.1 Basic ideas

We are going to simulate the Internet. In particular, we are going to simulate the *Google Pagerank* algorithm by which Google determines the importance of web pages.

Let's start with some basic classes:

- A *Page* contains some information such as its title and a global numbering in Google's data-center. It also contains a collection of links.
- We represent a link with a pointer to a *Page*. Conceivably we could have a *Link* class, containing further information such as probability of being clicked, or number of times clicked, but for now a pointer will do.
- Ultimately we want to have a class *Web* which contains a number of pages and their links. The *web* object will ultimately also contain information such as relative importance of the pages.

This application is a natural one for using pointers. When you click on a link on a web page you go from looking at one page in your browser to looking at another. You could implement this by having a pointer to a page, and clicking updates the value of this pointer.

**Exercise 50.1.** Make a class *Page* which initially just contains the name of the page. Write a method to display the page. Since we will be using pointers quite a bit, let this be the intended code for testing:

```
// google/web2.cpp
auto homepage = make_shared<Page>("My Home Page");
cout << "Homepage has no links yet:" << '\n';
cout << homepage->as_string() << '\n';
```

Next, add links to the page. A link is a pointer to another page, and since there can be any number of them, you will need a *vector* of them. Write a method *click* that follows the link. Intended code:

```
// google/web2.cpp
auto utexas = make_shared<Page>("University Home Page");
homepage->add_link(utexas);
auto searchpage = make_shared<Page>("google");
homepage->add_link(searchpage);
cout << homepage->as_string() << '\n';
```

**Exercise 50.2.** Add some more links to your homepage. Write a method `random_click` for the `Page` class. Intended code:

```
// google/web2.cpp
for (int iclick=0; iclick<20; ++iclick) {
    auto newpage = homepage->random_click();
    cout << "To: " << newpage->as_string() << '\n';
}
```

How do you handle the case of a page without links?

## 50.2 Clicking around

**Exercise 50.3.** Now make a class `Web` which foremost contains a bunch (technically: a `vector`) of pages. Or rather: of pointers to pages. Since we don't want to build a whole internet by hand, let's have a method `create_random_links` which makes a random number of links to random pages. Intended code:

```
// google/web2.cpp
Web internet(netsize);
internet.create_random_links(avglinks);
```

Now we can start our simulation. Write a method `Web::random_walk` that takes a page, and the length of the walk, and simulates the result of randomly clicking that many times on the current page. (Current page. Not the starting page.)

Let's start working towards PageRank. First we see if there are pages that are more popular than others. You can do that by starting a random walk once on each page. Or maybe a couple of times.

**Exercise 50.4.** Apart from the size of your internet, what other design parameters are there for your tests? Can you give a back-of-the-envelope estimation of their effect?

**Exercise 50.5.** Your first simulation is to start on each page a number of times, and counts where that lands you. Intended code:

```
// google/web2.cpp
vector<int> landing_counts(internet.number_of_pages(), 0);
for (auto page : internet.all_pages() ) {
    for (int iwalk=0; iwalk<5; ++iwalk) {
        auto endpage = internet.random_walk(page, 2*avglinks, tracing);
        landing_counts.at(endpage->global_ID())++;
    }
}
```

Display the results and analyze. You may find that you finish on certain pages too many times. What's happening? Fix that.

## 50.3 Graph algorithms

There are many algorithms that rely on gradually traversing the web. For instance, any graph can be *connected*. You test that by

- Take an arbitrary vertex  $v$ . Make a ‘reachable set’  $R \leftarrow \{v\}$ .
- Now see where you can get from your reachable set:

$$\forall_{v \in V} \forall_w \text{neighbour of } v : R \leftarrow R \cup \{w\}$$

- Repeat the previous step until  $R$  does not change anymore.

After this algorithm concludes, is  $R$  equal to your set of vertices? If so, your graph is called (fully) connected. If not, your graph has multiple *connected components*.

**Exercise 50.6.** Code the above algorithm, keeping track of how many steps it takes to reach each vertex  $w$ . This is the *Single Source Shortest Path* algorithm (for unweighted graphs).

The *diameter* is defined as the maximal shortest path. Code this.

## 50.4 Page ranking

The Pagerank algorithm now asks, if you keep clicking randomly, what is the distribution of how likely you are to wind up on a certain page. The way we calculate that is with a probability distribution: we assign a probability to each page so that the sum of all probabilities is one. We start with a random distribution:

**Code:**

```
// google/web2.cpp
ProbabilityDistribution
    random_state(internet.number_of_pages());
    random_state.set_random();
    cout << "Initial distribution: " <<
        random_state.as_string() << '\n';
```

**Output:**

```
[google] pdfsetup:
Initial distribution:
→0:0.00, 1:0.02, 2:0.07,
→3:0.05, 4:0.06, 5:0.08,
→6:0.04, 7:0.04, 8:0.04,
→9:0.01, 10:0.07,
→11:0.05, 12:0.01,
→13:0.04, 14:0.08,
→15:0.06, 16:0.10,
→17:0.06, 18:0.11,
→19:0.01,
```

**Exercise 50.7.** Implement a class *ProbabilityDistribution*, which stores a vector of floating point numbers. Write methods for:

- accessing a specific element,
- setting the whole distribution to random, and
- normalizing so that the sum of the probabilities is 1.
- a method rendering the distribution as string could be useful too.

Next we need a method that given a probability distribution, gives you the new distribution corresponding to performing a single click. (This is related to *Markov chains*; see HPC book [13], section 10.2.1.)

**Exercise 50.8.** Write the method

```
ProbabilityDistribution Web::globalclick  
    ( ProbabilityDistribution currentstate );
```

Test it by

- start with a distribution that is nonzero in exactly one page;
- print the new distribution corresponding to one click;
- do this for several pages and inspect the result visually.

Then start with a random distribution and run a couple of iterations. How fast does the process converge? Compare the result to the random walk exercise above.

**Exercise 50.9.** In the random walk exercise you had to deal with the fact that some pages have no outgoing links. In that case you transitioned to a random page. That mechanism is lacking in the `globalclick` method. Figure out a way to incorporate this.

Let's simulate some simple 'search engine optimization' trick.

**Exercise 50.10.** Add a page that you will artificially made look important: add a number of pages that all link to this page, but no one links to them. (Because of the random clicking they will still sometimes be reached.)

Compute the rank of the artificially hyped page. Did you manage to trick Google into ranking this page high? How many links did you have to add?

Sample output:

```
Internet has 5000 pages  
Top score: 109:0.0013, 3179:0.0012, 4655:0.0010, 3465:0.0009, 4298:0.0008  
With fake pages:  
Internet has 5051 pages  
Top score: 109:0.0013, 3179:0.0012, 4655:0.0010, 5050:0.0010, 4298:0.0008  
Hyped page scores at 4
```

## 50.5 Graphs and linear algebra

The probability distribution is essentially a vector. You can also represent the web as a matrix  $W$  with  $w_{ij} = 1$  if page  $i$  links to page  $j$ . (For details see HPC book [13], section 25.5.) How can you interpret the `globalclick` method in these terms?

**Exercise 50.11.** Add the matrix representation of the `Web` object and reimplement the `globalclick` method. Test for correctness.

Do a timing comparison.

The iteration you did above to find a stable probability distribution corresponds to the 'power method' in linear algebra. Look up the Perron-Frobenius theory and see what it implies for page ranking.

## 50.6 Bonus: parallelism

Explore the OpenMP library to parallelize the various algorithms you have developed in this project.

Argue that a Compressed Row Storage (CRS) sparse matrix representation is easy to parallelize. How is that for the more direct implementation of a graph as a linked structure?



# Chapter 51

## Redistricting

In this project you can explore ‘gerrymandering’, the strategic drawing of districts to give a minority population a majority of districts<sup>1</sup>.

### 51.1 Basic concepts

We are dealing with the following concepts:

- A state is divided into census districts, which are given. Based on census data (income, ethnicity, median age) one can usually make a good guess as to the overall voting in such a district.
- There is a predetermined number of congressional districts, each of which consists of census districts. A congressional district is not a random collection: the census districts have to be contiguous.
- Every couple of years, to account for changing populations, the district boundaries are redrawn. This is known as redistricting.

There is considerable freedom in how redistricting is done: by shifting the boundaries of the (congressional) districts it is possible to give a population that is in the overall minority a majority of districts. This is known as *gerrymandering* [22].

For background reading, see <https://redistrictingonline.org/>.

To do a small-scale computer simulation of gerrymandering, we make some simplifying assumption.

- First of all, we dispense with census district: we assume that a district consists directly of voters, and that we know their affiliation. In practice one relies on proxy measures (such as income and education level) to predict affiliation.
- Next, we assume a one-dimensional state. This is enough to construct examples that bring out the essence of the problem:

Consider a state of five voters, and we designate their votes as AAABB. Assigning them to three (contiguous) districts can be done as AAA | B | B, which has one ‘A’ district and two ‘B’ districts.

- We also allow districts to be any positive size, as long as the number of districts is fixed.

---

1. This project is obviously based on the Northern American political system. Hopefully the explanations here are clear enough. Please contact the author if you know of other countries that have a similar system.

## 51.2 Basic functions

### 51.2.1 Voters

We dispense with census districts, expressing everything in terms of voters, for which we assume a known voting behavior. Hence, we need a *Voter* class, which will record the voter ID and party affiliation. We assume two parties, and leave an option for being undecided.

**Exercise 51.1.** Implement a *Voter* class. You could for instance let  $\pm 1$  stand for A/B, and 0 for undecided.

```
// gerry/linear.cpp
cout << "Voter 5 is positive:" << '\n';
Voter nr5(5,+1);
cout << nr5.print() << '\n';

// gerry/linear.cpp
cout << "Voter 6 is negative:" << '\n';
Voter nr6(6,-1);
cout << nr6.print() << '\n';

// gerry/linear.cpp
cout << "Voter 7 is weird:" << '\n';
Voter nr7(7,3);
cout << nr7.print() << '\n';
```

### 51.2.2 Populations

**Exercise 51.2.** Implement a *District* class that models a group of voters.

- You probably want to create a district out of a single voter, or a vector of them. Having a constructor that accepts a string representation would be nice too.
- Write methods *majority* to give the exact majority or minority, and *lean* that evaluates whether the district overall counts as A party or B party.
- Write a *sub* method to creates subsets.

```
District District::sub(int first,int last);
```

- For debugging and reporting it may be a good idea to have a method

```
string District::print();
```

**Code:**

```
// gerry/linear.cpp
cout << "Making district with one B voter"
    << '\n';
Voter nr5(5,+1);
District nine( nr5 );
cout << "... size: " << nine.size() << '\n'
';
cout << "... lean: " << nine.lean() << '\n'
';
/* ... */
cout << "Making district ABA" << '\n';
District nine( vector<Voter>
                { {1,-1},{2,+1},{3,-1} } );
cout << "... size: " << nine.size() << '\n'
';
cout << "... lean: " << nine.lean() << '\n'
';
```

**Output:**

```
[gerry] district:
Making district with one B voter
.. size: 1
.. lean: 1

Making district ABA
.. size: 3
.. lean: -1
```

**Exercise 51.3.** Implement a *Population* class that will initially model a whole state.**Code:**

```
// gerry/linear.cpp
string pns( "-++--" );
Population some(pns);
cout << "Population from string "
    << pns << '\n';
cout << "... size: " << some.size() << '\n';
cout << "... lean: " << some.lean() << '\n';
Population group=some.sub(1,3);
cout << "sub population 1--3" << '\n';
cout << "... size: " << group.size() << '\n';
cout << "... lean: " << group.lean() << '\n';
```

**Output:**

```
[gerry] population:
Population from string -+-
.. size: 5
.. lean: -1
sub population 1--3
.. size: 2
.. lean: 1
```

In addition to an explicit creation, also write a constructor that specifies how many people and what the majority is:

```
// gerry/linear.cpp
Population( int population_size, int majority, bool trace=false )
```

Use a random number generator to achieve precisely the indicated majority.

### 51.2.3 Districting

The next level of complication is to have a set of districts. Since we will be creating this incrementally, we need some methods for extending it.

**Exercise 51.4.** Write a class *Districting* that stores a *vector* of *District* objects. Write *size* and *lean* methods:

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                                                                                                                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Code:</b><br><pre>// gerry/linear.cpp cout &lt;&lt; "Making single voter population B" &lt;&lt;     '\n'; Population people( vector&lt;Voter&gt;{ Voter(0,+1) } ); cout &lt;&lt; "... size: " &lt;&lt; people.size() &lt;&lt; '\n'; cout &lt;&lt; "... lean: " &lt;&lt; people.lean() &lt;&lt; '\n';  Districting gerry; cout &lt;&lt; "Start with empty districting:" &lt;&lt; '\n'     ; cout &lt;&lt; "... number of districts: "     &lt;&lt; gerry.size() &lt;&lt; '\n';</pre> | <b>Output:</b><br><pre>[gerry] gerryempty: Making single voter population B .. size: 1 .. lean: 1 Start with empty districting: .. number of districts: 0</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Exercise 51.5. Write methods to extend a *Districting*:

```
// gerry/linear.cpp
cout << "Add one B voter:" << '\n';
gerry = gerry.extend_with_new_district( people.at(0) );
cout << "... number of districts: " << gerry.size() << '\n';
cout << "... lean: " << gerry.lean() << '\n';
cout << "add A A:" << '\n';
gerry = gerry.extend_last_district( Voter(1,-1) );
gerry = gerry.extend_last_district( Voter(2,-1) );
cout << "... number of districts: " << gerry.size() << '\n';
cout << "... lean: " << gerry.lean() << '\n';

cout << "Add two B districts:" << '\n';
gerry = gerry.extend_with_new_district( Voter(3,+1) );
gerry = gerry.extend_with_new_district( Voter(4,+1) );
cout << "... number of districts: " << gerry.size() << '\n';
cout << "... lean: " << gerry.lean() << '\n';
```

## 51.3 Strategy

Now we need a method for districting a population:

```
Districting Population::minority_rules( int ndistricts );
```

Rather than generating all possible partitions of the population, we take an incremental approach (this is related to the solution strategy called *dynamic programming*):

- The basic question is to divide a population optimally over  $n$  districts;
- We do this recursively by first solving a division of a subpopulation over  $n - 1$  districts,
- and extending that with the remaining population as one district.

This means that you need to consider all the ways of having the ‘remaining’ population into one district, and that means that you will have a loop over all ways of splitting the population, outside of your recursion; see figure 51.1.

- For all  $p = 0, \dots, n - 1$  considering splitting the state into  $0, \dots, p - 1$  and  $p, \dots, n - 1$ .
- Use the best districting of the first group, and make the last group into a single district.

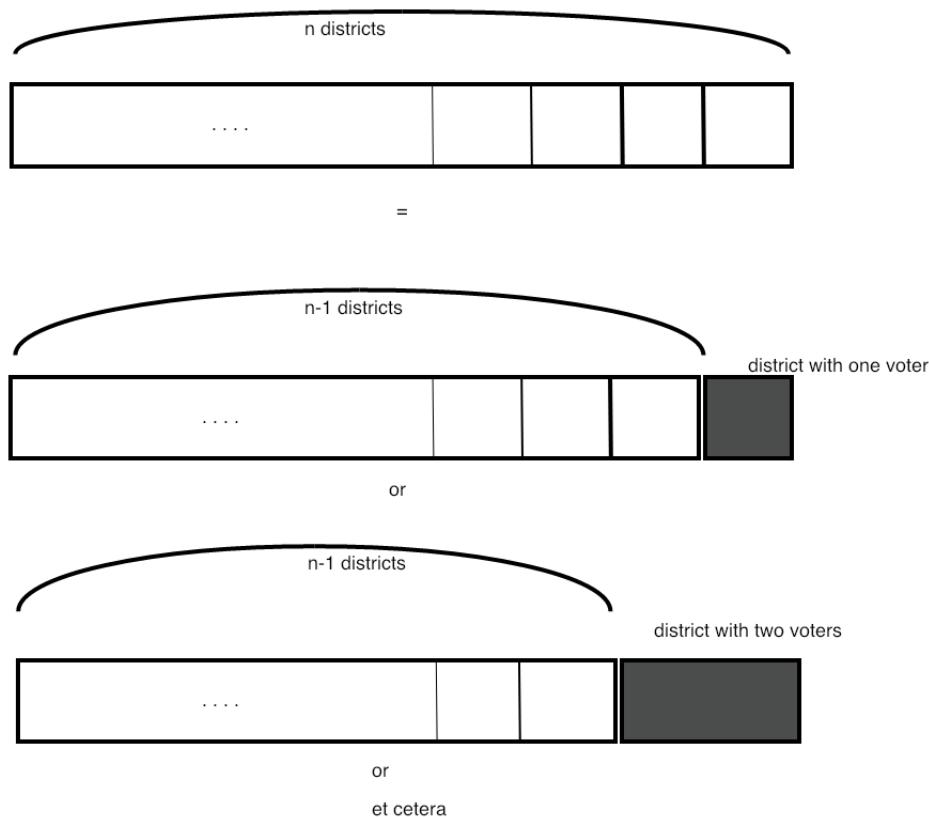


Figure 51.1: Multiple ways of splitting a population

- Keep the districting that gives the strongest minority rule, over all values of  $p$ .

You can now realize the above simple example:

AAABB => AAA | B | B

**Exercise 51.6.** Implement the above scheme.

**Code:**

```
// gerry/linear.cpp
Population five("++++-");
cout << "Redistricting population: " << '\n'
    << five.print() << '\n';
cout << "... majority rule: "
    << five.rule() << '\n';
int ndistricts{3};
auto gerry = five.minority_rules(ndistricts);
cout << gerry.print() << '\n';
cout << "... minority rule: "
    << gerry.rule() << '\n';
```

**Output:**

```
[gerry] district5:
Redistricting population:
[0:+,1:+,2:+,3:-,4:-,]
.. majority rule: 1
[3[0:+,1:+,2:+,], [3:-,], [4:-,], ]
.. minority rule: -1
```

Note: the range for  $p$  given above is not quite correct: for instance, the initial part of the population needs to be big enough to accommodate  $n - 1$  voters.

**Exercise 51.7.** Test multiple population sizes; how much majority can you give party B while still giving party A a majority.

### 51.4 Efficiency: dynamic programming

If you think about the algorithm you just implemented, you may notice that the districtings of the initial parts get recomputed quite a bit. A strategy for optimizing for this is called *memoization*.

**Exercise 51.8.** Improve your implementation by storing and reusing results for the initial sub-populations.

In a way, we solved the program backward: we looked at making a district out of the last so-many voters, and then recursively solving a smaller problem for the first however-many voters. But in that process, we decided what is the best way to assign districts to the first 1 voter, first 2, first 3, et cetera. Actually, for more than one voter, say five voters, we found the result on the best attainable minority rule assigning these five voters to one, two, three, four districts.

The process of computing the ‘best’ districting forward, is known as *dynamic programming*. The fundamental assumption here is that you can use intermediate results and extend them, without having to reconsider the earlier problems.

Consider for instance that you’ve considered districting ten voters over up to five districts. Now the majority for eleven voters and five districts is the minimum of

- ten voters and five districts, and the new voter is added to the last district; or
- ten voters and four districts, and the new voter becomes a new district.

**Exercise 51.9.** Code a dynamic programming solution to the redistricting problem.

### 51.5 Extensions

The project so far has several simplifying assumptions.

- Congressional districts need to be approximately the same size. Can you put a limit on the ratio between sizes? Can the minority still gain a majority?

**Exercise 51.10.** The biggest assumption is of course that we considered a one-dimensional state. With two dimensions you have more degrees of freedom of shaping the districts. Implement a two-dimensional scheme; use a completely square state, where the census districts form a regular grid. Limit the shape of the congressional districts to be convex.

The *efficiency gap* is a measure of how ‘fair’ a districting of a state is.

**Exercise 51.11.** Look up the definition of efficiency gap (and ‘wasted votes’), and implement it in your code.

## 51.6      Ethics

The activity of redistricting was intended to give people a fair representation. In its degenerate form of Gerrymandering this concept of fairness is violated because the explicit goal is to give the minority a majority of votes. Explore ways that this unfairness can be undone.

In your explorations above, the only characteristic of a voter was their preference for party A or B. However, in practice voters can be considered part of communities. The Voting Rights Act is concerned about ‘minority vote dilution’. Can you show examples that a color-blind districting would affect some communities negatively?



# Chapter 52

## Amazon delivery truck scheduling

This section contains a sequence of exercises that builds up to a simulation of delivery truck scheduling.

### 52.1 Problem statement

Scheduling the route of a delivery truck is a well-studied problem. For instance, minimizing the total distance that the truck has to travel corresponds to the *Traveling Salesman Problem (TSP)*. However, in the case of *Amazon delivery truck* scheduling the problem has some new aspects:

- A customer is promised a window of days when delivery can take place. Thus, the truck can split the list of places into sublists, with a shorter total distance than going through the list in one sweep.
- Except that *Amazon prime* customers need their deliveries guaranteed the next day.

### 52.2 Coding up the basics

Before we try finding the best route, let's put the basics in place to have any sort of route at all.

#### 52.2.1 Address list

You probably need a class `Address` that describes the location of a house where a delivery has to be made.

- For simplicity, let give a house  $(i, j)$  coordinates.
- We probably need a `distance` function between two addresses. We can either assume that we can travel in a straight line between two houses, or that the city is build on a grid, and you can apply the so-called *Manhattan distance*.
- The address may also require a field recording the last possible delivery date.

**Exercise 52.1.** Code a class `Address` with the above functionality, and test it.



**Code:**

```
// amazon/route.cpp
Address one(1.,1.),
two(2.,2.);
cerr << "Distance: "
<< one.distance(two)
<< '\n';
```

**Output:**

[amazon] address:  
Address  
Distance: 1.41421  
.. address  
Address 1 should be closest to  
→the depot. Check: 1

Route from depot to depot:  
→(0,0) (2,0) (1,0) (3,0)  
→(0,0)  
has length 8: 8  
Greedy scheduling: (0,0) (1,0)  
→(2,0) (3,0) (0,0)  
should have length 6: 6

Square5  
Travel in order: 24.1421  
Square route: (0,0) (0,5)  
→(5,5) (5,0) (0,0)  
has length 20  
.. square5

Original list: (0,0) (-2,0)  
→(-1,0) (1,0) (2,0) (0,0)  
length=8  
flip middle two addresses:  
→(0,0) (-2,0) (1,0) (-1,0)  
→(2,0) (0,0)  
length=12  
better: (0,0) (1,0) (-2,0)  
→(-1,0) (2,0) (0,0)  
length=10

Hundred houses  
Route in order has length  
→25852.6  
TSP based on mere listing has  
→length: 2751.99 over  
→naive 25852.6  
Single route has length: 2078.43  
.. new route accepted with  
→length 2076.65  
Final route has length 2076.65  
→over initial 2078.43  
TSP route has length 1899.4  
→over initial 2078.43

Two routes  
Route1: (0,0) (2,0) (3,2)  
→(2,3) (0,2) (0,0)  
route2: (0,0) (3,1) (2,1)  
→(1,2) (1,3) (0,0)  
total length 19.6251  
start with 9.88635, 9.73877  
Pass 0  
.. down to 9.81256, 8.57649  
Pass 1  
Pass 2  
Pass 3  
Pass 4  
TSP Route1: (0,0) (3,1) (3,2)  
→(2,3) (0,2) (0,0)  
route2: (0,0) (2,0) (2,1)  
→(1,2) (1,3) (0,0)  
total length 18.389

Next we need a class `AddressList` that contains a list of addresses.

**Exercise 52.2.** Implement a class `AddressList`; it probably needs the following methods:

- `add_address` for constructing the list;
- `length` to give the distance one has to travel to visit all addresses in order;
- `index_closest_to` that gives you the address on the list closest to another address, presumably not on the list.

You have options here.

- Do you let your `AddressList` actually contain a `vector<Address>` or do you let it inherit from that container? See section 10.7.
- Another approach is to use a linked list. This has certain advantages; see section 52.6.

Write your code sufficiently abstract that you can accomodate these options.

### 52.2.2 Add a depot

Next, we model the fact that the route needs to start and end at the depot, which we put arbitrarily at coordinates (0, 0). We could construct an `AddressList` that has the depot as first and last element, but that may run into problems:

- If we reorder the list to minimize the driving distance, the first and last elements may not stay in place.
- We may want elements of a list to be unique: having an address twice means two deliveries at the same address, so the `add_address` method would check that an address is not already in the list.

We can solve this by making a class `Route`, which inherits from `AddressList`, but the methods of which leave the first and last element of the list in place.

### 52.2.3 Greedy construction of a route

Next we need to construct a route. Rather than solving the full TSP, we start by employing a *greedy search* strategy:

Given a point, find the next point by some local optimality test, such as shortest distance. Never look back to revisit the route you have constructed so far.

Such a strategy is likely to give an improvement, but most likely will not give the optimal route.

Let's write a method

```
Route::Route greedy_route();
```

that constructs a new address list, containing the same addresses, but arranged to give a shorter length to travel.

**Exercise 52.3.** Write the `greedy_route` method for the `AddressList` class.

1. Assume that the route starts at the depot, which is located at (0, 0). Then incrementally construct a new list by:
2. Maintain an `Address` variable `we_are_here` of the current location;

3. repeatedly find the address closest to `we_are_here`.

Extend this to a method for the `Route` class by working on the subvector that does not contain the final element.

Test it on this example:

**Code:**

```
// amazon/route.cpp
Route deliveries;
deliveries.add_address( Address(0,5) );
deliveries.add_address( Address(5,0) );
deliveries.add_address( Address(5,5) );
cerr << "Travel in order: "
     << deliveries.length() << '\n';
assert( deliveries.size()==5 );
auto route = deliveries.greedy_route();
assert( route.size()==5 );
auto len = route.length();
cerr << "Square route:\n"
     << route.as_string()
     << "\n has length " << len << '\n';
```

**Output:**

[amazon] **square5:**

Travel in order: 24.1421  
 Square route: (0,0) (0,5)  
 $\rightarrow$ (5,5) (5,0) (0,0)  
 has length 20

Reorganizing a list can be done in a number of ways.

- First of all, you can try to make the changes in place. This runs into the objection that maybe you want to save the original list; also, while swapping two elements can be done with the `insert` and `erase` methods, more complicated operations are tricky.
- Alternatively, you can incrementally construct a new list. Now the main problem is to keep track of which elements of the original have been processed. You could do this by giving each address a boolean field `done`, but you could also make a copy of the input list, and remove the elements that have been processed. For this, study the `erase` method for `vector` objects.

## 52.3 Optimizing the route

The above suggestion of each time finding the closest address is known as a *greedy search* strategy. It does not give you the optimal solution of the TSP. Finding the optimal solution of the TSP is hard to program – you could do it recursively – and takes a lot of time as the number of addresses grows. In fact, the TSP is probably the most famous of the class of *NP-hard* problems, which are generally believed to have a running time that grows faster than polynomial in the problem size.

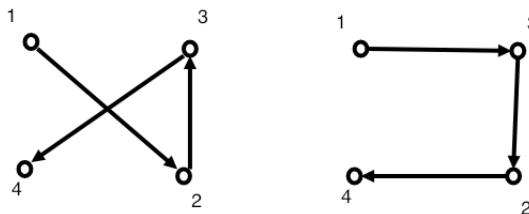


Figure 52.1: Illustration of the ‘Opt2’ idea of reversing part of a path

However, you can approximate the solution heuristically. One method, the Kernighan-Lin algorithm [25], is based on the *opt2* idea: if you have a path that ‘crosses itself’, you can make it shorter by reversing part of it. Figure 52.1 shows that the path  $1 - 2 - 3 - 4$  can be made shorter by reversing part of it, giving  $1 - 3 - 2 - 4$ . Since recognizing where a path crosses itself can be hard, or even impossible for graphs that don’t have Cartesian coordinates associated, we adopt a scheme where we try all possible reversals:

```
for all nodes m < n on the path [1..N] :  
    make a new route from  
        [1..m-1] + [m--n].reversed + [n+1..N]  
    if the new route is shorter, keep it
```

**Exercise 52.4.** Code the Opt2 heuristic: write a method to reverse part of the route, and write the loop that tries this with multiple starting and ending points. Try it out on some simple test cases to convince you that your code works as intended.

Let’s explore issues of complexity. (For an introduction to complexity calculations, see HPC book [13], section 20.) The TSP is one of a class of *NP complete* problems, which very informally means that there is no better solution than trying out all possibilities.

**Exercise 52.5.** What is the runtime complexity of the heuristic solution using Opt2? What would the runtime complexity be of finding the best solution by considering all possibilities? Make a very rough estimation of runtimes of the two strategies on some problem sizes:  $N = 10, 100, 1000, \dots$

**Exercise 52.6.** (Optional) Reversing sublists is an important operation on your data structure. How many operations does that take? Explore implementing your *AddressList* not as a *vector* but as a *std::list*: a linked list. What is the complexity of reversing a sublist now? Do timing measurements bear this out?

**Exercise 52.7.** Earlier you had programmed the greedy heuristic. Compare the improvement you get from the Opt2 heuristic, starting both with the given list of addresses, and with a greedy traversal of it. For realism, how many addresses do you put on your route? How many addresses would a delivery driver do on a typical day?

## 52.4 Multiple trucks

If we introduce multiple delivery trucks, we get the ‘Multiple Traveling Salesman Problem’ [5]. With this we can module both the cases of multiple trucks being out on delivery on the same day, or one truck spreading deliveries over multiple days. For now we don’t distinguish between the two.

The first question is how to divide up the addresses.

1. We could split the list in two, using some geometric test. This is a good model for the case where multiple trucks are out on the same day. However, if we use this as a model for the same truck being out on multiple days, we are missing the fact that new addresses can be added on the first day, messing up the neatly separated routes.

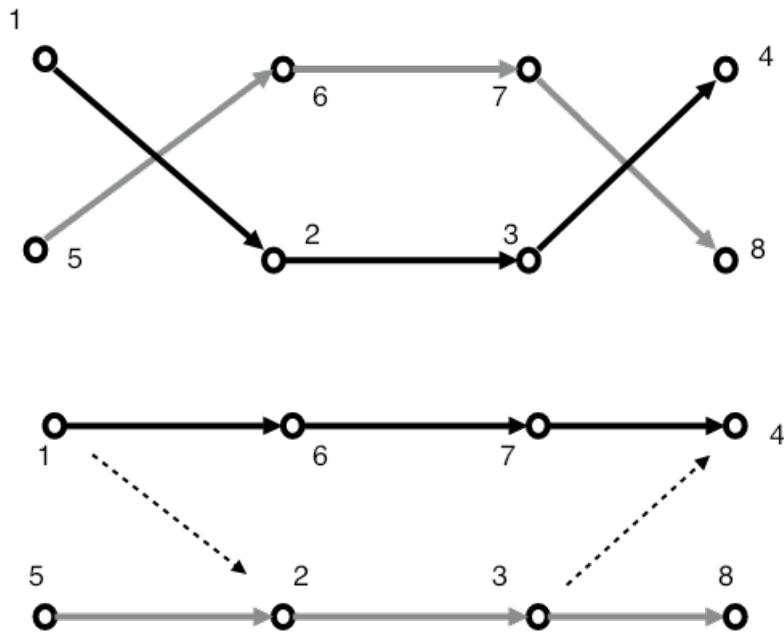


Figure 52.2: Extending the ‘opt2’ idea to multiple paths

2. Thus it may in fact be reasonable to assume that all trucks get an essentially random list of addresses.

Can we extend the Opt2 heuristic to the case of multiple paths? For inspiration take a look at figure 52.2: instead of modifying one path, we could switch bits out bits between one path and another. When you write the code, take into account that the other path may be running backwards! This means that based on split points in the first and second path you know have four resulting modified paths to consider.

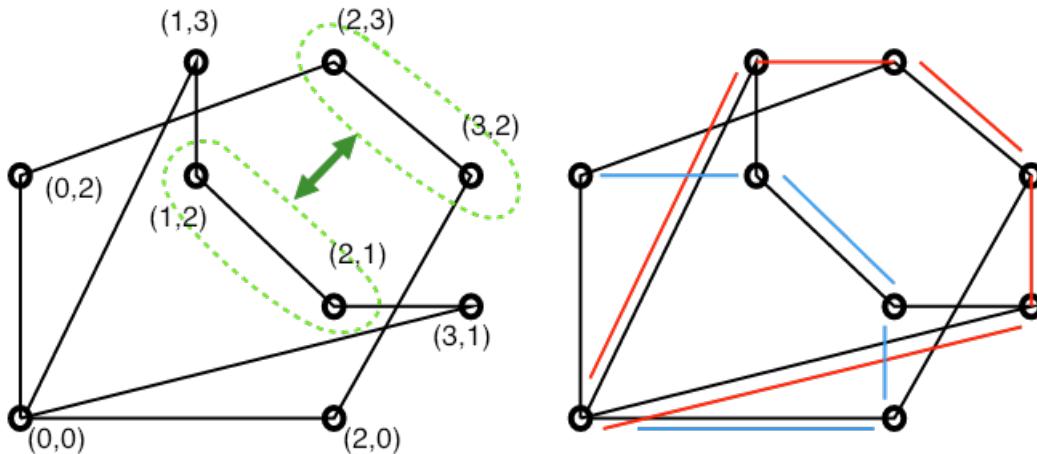


Figure 52.3: Multiple paths test case

**Exercise 52.8.** Write a function that optimizes two paths simultaneously using the multi-path version of the Opt2 heuristic. For a test case, see figure 52.3.

You have quite a bit of freedom here:

- The start points of the two segments should be chosen independently;
- the lengths can be chosen independently, but need not; and finally
- each segment can be reversed.

More flexibility also means a longer runtime of your program. Does it pay off? Do some tests and report results.

Based on the above description there will be a lot of code duplication. Make sure to introduce functions and methods for various operations.

## 52.5 Amazon prime

In section 52.4 you made the assumption that it doesn't matter on what day a package is delivered. This changes with *Amazon prime*, where a package has to be delivered guaranteed on the next day.

**Exercise 52.9.** Explore a scenario where there are two trucks, and each have a number of addresses that can not be exchanged with the other route. How much longer is the total distance? Experiment with the ratio of prime to non-prime addresses.

## 52.6 Dynamicism

So far we have assumed that the list of addresses to be delivered to is given. This is of course not true: new deliveries will need to be scheduled continuously.

**Exercise 52.10.** Implement a scenario where every day a random number of new deliveries is added to the list. Explore strategies and design choices.

## 52.7 Bonus: parallelism

In this project you implemented two algorithms: the greedy strategy, and the Opt2 heuristic. The latter is the easier to parallelize: the Opt2 code explores many possibilities of where to exchange route segments. Implement this with the OpenMP tasking mechanism; MPI/OpenMP book [12], section 24.

The greedy strategy is harder to parallelize. Can you come up with a strategy that is parallel, but may not give quite the same result?

## 52.8 Ethics

People sometimes criticize Amazon's labor policies, including regarding its drivers. Can you make any observations from your simulations in this respect?

## Chapter 53

### High performance linear algebra

Linear algebra is fundamental to much of computational science. Applications involving Partial Differential Equations (PDEs) come down to solving large systems of linear equation; solid state physics involves large eigenvalue systems. But even outside of engineering applications linear algebra is important: the major computational part of Deep Learning (DL) networks involves matrix-matrix multiplications.

Linear algebra operations such as the matrix-matrix product are easy to code in a naive way. However, this does not lead to high performance. In these exercises you will explore the basics of a strategy for high performance.

**Remark** The algorithm you will develop in this project is an example of cache-oblivious programming; see HPC book [13], section 6.9. While this gives considerably higher performance than the naive algorithm, for the highest performance a more sophisticated approach is needed, which involves considerable tuning, as well as a little assembly coding [17].

#### 53.1 Mathematical preliminaries

The matrix-matrix product  $C \leftarrow A \cdot B$  is defined as

$$\forall_{ij} : c_{ij} \leftarrow \sum_k a_{ik} b_{kj}.$$

Straightforward code for this would be:

```
for (i=0; i<a.m; i++)
    for (j=0; j<b.n; j++)
        s = 0;
        for (k=0; k<a.n; k++)
            s += a[i, k] * b[k, j];
        c[i, j] = s;
```

However, this is not the only way to code this operation. The loops can be permuted, giving a total of six implementations.

**Exercise 53.1.** Code one of the permuted algorithms and test its correctness. If the reference algorithm above can be said to be ‘inner-product based’, how would you describe your variant?

Yet another implementation is based on a block partitioning. Let  $A, B, C$  be split on  $2 \times 2$  block form:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{53.1}$$

Convince yourself that this actually computes the same product  $C = A \cdot B$ . For more on block algorithms, see HPC book [13], section 5.3.6.

**Exercise 53.2.** Write a matrix class with a multiplication routine:

```
Matrix Matrix::MatMult( Matrix other );
```

First implement a traditional matrix-matrix multiplication, then make it recursive. For the recursive algorithm you need to implement sub-matrix handling: you need to extract submatrices, and write a submatrix back into the surrounding matrix.

Note: this copying is not efficient. Later you will see a better strategy.

## 53.2 Matrix storage

The simplest way to store an  $M \times N$  matrix is as an array of length  $MN$ . Inside this array we can decide to store the rows end-to-end, or the columns. While this decision is obviously of practical importance for a library, from a point of performance it makes no difference.

**Remark** Historically, linear algebra software such as the BLAS has used columnwise storage, meaning that the location of an element  $(i, j)$  is computed as  $i + j \cdot M$  (we will use zero-based indexing throughout this project, both for code and mathematical expressions.) The reason for this stems from the origins of the BLAS in the Fortran language, which uses column-major ordering of array elements. On the other hand, C-style arrays (such as `x[5][6]`) in the C/C++ languages have row-major ordering, where element  $(i, j)$  is stored in location  $j + i \cdot N$ .

Above, you saw the idea of block algorithms, which requires taking submatrices. For efficiency, we don't want to copy elements into a new array, so we want the submatrix to correspond to a subarray.

Now we have a problem: only a submatrix that consists of a sequence of columns is contiguous. The formula  $i + j \cdot M$  for location of element  $(i, j)$  is no longer correct if the matrix is a subblock of a larger matrix.

For this reason, linear algebra software describes a submatrix by three parameters  $M, N, LDA$ , where 'LDA' stands for 'leading dimension of  $A$ ' (see BLAS [21], and Lapack [1]). This is illustrated in figure 53.1.

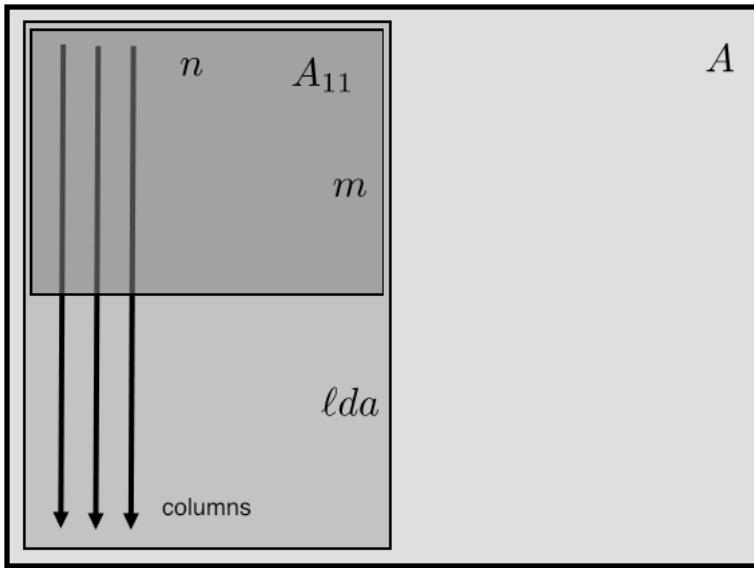


Figure 53.1: Submatrix out of a matrix, with  $M, N, LDA$  of the submatrix indicated

**Exercise 53.3.** In terms of  $M, N, LDA$ , what is the location of the  $(i, j)$  element?

Implementationwise we also have a problem. If we use `std::vector` for storage, it is not possible to take subarrays, since C++ insists that a vector has its own storage. The solution is to use `span`; section 10.8.3.

Now we have types matrices: top level matrices that store a `vector<double>`, and submatrices that store a `span<double>`. It could be done using `std::variant` (section 24.6.4), but let's not.

Instead, let's adopt the following idiom, where we create a vector at the top level, and then create matrices from its memory.

```
// gemm/lapack_alloc.cpp
// example values for M,LDA,N
M = 2; LDA = M+2; N = 3;
// create a vector to contain the data
vector<double> one_data(LDA*N, 1.);
// create a matrix using the vector data
Matrix one(M, LDA, N, one_data.data());
```

Now all types of matrices are created from a `double*`. (If you have not previously programmed in C, you need to get used to the `double*` mechanism. See section 10.10.)

**Exercise 53.4.** Start implementing the `Matrix` class with a constructor

```
Matrix::Matrix(int m, int lda, int n, double *data)
```

and private data members:

```
// gemm/lapack_alloc.cpp
private:
    int m, n, lda;
    span<double> data;
```

Write a method

```
double& Matrix::at(int i,int j);
```

that you can use as a safe way of accessing elements.

**Exercise 53.5.** Do the above exercise, using the C++23 `msdspan` construct.

Let's start with simple operations.

**Exercise 53.6.** Write a method for adding matrices. Test it on matrices that have the same  $M, N$ , but different  $LDA$ .

Use of the `at` method is great for debugging, but it is not efficient. Use the preprocessor (chapter 21) to introduce alternatives:

```
#ifdef DEBUG
    c.at(i,j) += a.at(i,k) * b.at(k,j)
#else
    cdata[ /* expression with i,j */ ] += adata[ ... ] * bdata[ ... ]
#endif
```

where you access the data directly with

```
// gemm/lapack_alloc.cpp
auto get_double_data() {
    double *adata;
    adata = data.data();
    return adata;
};
```

**Exercise 53.7.** Implement this. Use the `#define` preprocessor directive for the optimized indexing expression. (See section 21.2.2.)

### 53.2.1 Submatrices

Next we need to support constructing actual submatrices. Since we will mostly aim for decomposition in  $2 \times 2$  block form, it is enough to write four methods. This can actually be done two different ways:

|                                   |                                            |
|-----------------------------------|--------------------------------------------|
| <code>Matrix Left(int j);</code>  | <code>Matrix TopLeft(int i,int j);</code>  |
| <code>Matrix Right(int j);</code> | <code>Matrix TopRight(int i,int j);</code> |
| <code>Matrix Top(int i);</code>   | <code>Matrix BotLeft(int i,int j);</code>  |
| <code>Matrix Bot(int i);</code>   | <code>Matrix BotRight(int i,int j);</code> |

where, for instance, `Left(5)` gives the columns with  $j < 5$ , or `BotRight(5, 6)` give the bottom right block starting at 5, 6.

**Exercise 53.8.** Implement these methods and test them.

### 53.3 Multiplication

You can now write a first multiplication routine, for instance with a prototype

```
void Matrix::MatMult( Matrix& other, Matrix& out );
```

Alternatively, you could write

```
Matrix Matrix::MatMult( Matrix& other );
```

but we want to keep the amount of creation/destruction of objects to a minimum.

#### 53.3.1 One level of blocking

Next, write

```
void Matrix::BlockedMatMult( Matrix& other, Matrix& out );
```

which uses the  $2 \times 2$  form above.

#### 53.3.2 Recursive blocking

The final step is to make the blocking recursive.

**Exercise 53.9.** Write a method

```
void RecursiveMatMult( Matrix& other, Matrix& out );
```

which

- Executes the  $2 \times 2$  block product, using again `RecursiveMatMult` for the blocks.
- When the block is small enough, use the regular `MatMult` product.

### 53.4 Performance issues

If you experiment a little with the cutoff between the regular and recursive matrix-matrix product, you see that you can get good factor of performance improvement. Why is this?

The matrix-matrix product is a basic operation in scientific computations, and much effort has been put into optimizing it. One interesting fact is that it is just about the most optimizable operation under the sun. The reason for this is, in a nutshell, that it involves  $O(N^3)$  operations on  $O(N^2)$  data. This means that, in principle each element fetched will be used multiple times, thereby overcoming the *memory bottleneck*.

**Exercise 53.10.** Make sure that your code has timers; see chapter 24.8. Can you see a performance difference between the naive code and your optimized code? What is the peak speed of your processor, and how close are you getting to it? Is this number dependent on the size of your matrix?

To understand performance issues relating to hardware, you need to do some reading. Section HPC book [13], section 1.3.5 explains the crucial concept of a *cache*.

**Exercise 53.11.** Argue that the naive matrix-matrix product implementation is unlikely actually to reuse data.

Explain why the recursive strategy does lead to data reuse.

Above, you set a cutoff point for when to switch from the recursive to the regular product.

**Exercise 53.12.** Argue that continuing to recurse will not have much benefit once the product is contained in the cache. What are the cache sizes of your processor?

Do experiments with various cutoff points. Can you relate this to the cache sizes?

### 53.5 Bonus: parallelism

The four clauses of equation 53.1 target independent areas in the  $C$  matrix, so they could be executed in parallel on any processor that has at least four cores.

Use the ‘task’ mechanism of the OpenMP library to parallelize the `BlockedMatMult`.

### 53.6 Bonus: optimal performance

The final question is: how close are you getting to the best possible speed? Unfortunately you are still a way off. You can explore that as follows.

Your computer is likely to have an optimized implementation, accessible through:

```
#include <cblas.h>

cblas_dgemm
( CblasColMajor, CblasNoTrans, CblasNoTrans,
  m, other.n, n, alpha, adata, lda,
  bdata, other lda,
  beta, cdata, out.lda);
```

which computes  $C \leftarrow \alpha A \cdot B + \beta C$ .

**Exercise 53.13.** Use another cpp conditional to implement `MatMult` through a call to `cblas_dgemm`. What performance do you now get?

You see that your recursive implementation is faster than the naive one, but not nearly as fast as the CBlas one. This is because

- the CBlas implementation is probably based on an entirely different strategy [17], and
- it probably involves a certain amount of assembly coding.

## Chapter 54

### The Great Garbage Patch

This section contains a sequence of exercises that builds up to a *cellular automaton* simulation of turtles in the ocean, garbage that is deadly to them, and ships that clean it up. To read more about this: <https://theoceancleanup.com/>.

Thanks to Ernesto Lima of TACC for the idea and initial code of this exercise.

#### 54.1 Problem and model solution

There is lots of plastic floating around in the ocean, and that is harmful to fish and turtles and cetaceans. Here you get to model the interaction between

- Plastic, randomly located;
- Turtles that swim about; they breed slowly, and they die from ingesting plastic;
- Ships that sweep the ocean to remove plastic debris.

The simulation method we use is that of a *cellular automaton*:

- We have a grid of cells;
- Each cell has a ‘state’ associated with it, namely, it can contain a ship, a turtle, or plastic, or be empty; and
- On each next time step, the state of a cell is a simple function of the states of that cell and its immediate neighbors.

The purpose of this exercise is to simulate a number of time steps, and explore the interaction between parameters: with how much garbage will turtles die out, how many ships are enough to protect the turtles.

#### 54.2 Program design

The basic idea is to have an `ocean` object that is populated with turtles, trash, and ships. Your simulation will let the ocean undergo a number of time steps:

```
for (int t=0; t<time_steps; t++)
    ocean.update();
```

Ultimately your purpose is to investigate the development of the turtle population: is it stable, does it die out?

While you can make a ‘hackish’ solution to this problem, partly you will be judged on your use of modern/clean C++ programming techniques. A number of suggestions are made below.

### 54.2.1 Grid update

Here is a point to be aware of. Can you see what's wrong with doing an update entirely in-place:

```
for ( i )
    for ( j )
        cell(i, j) = f( cell(i, j), ... other cells ... );
?
```

## 54.3 Testing

It can be complicated to test this program for correctness. The best you can do is to try out a number of scenarios. For that it's best to make your program input flexible: use the `cxxopts` package [9], and drive your program from a shell script.

Here is a list of things you can test.

1. Start with only a number of ships; check that after 1000 time steps you still have the same number.
2. Likewise with turtles; if they don't breed and don't die, check that their number stays constant.
3. With only ships and trash, does it all get swept?
4. With only turtles and trash, do they all die off?

It is harder to test that your turtles and ships don't 'teleport' around, but only move to contiguous cells. For that, use visual inspection; see section 54.3.1.

### 54.3.1 Animated graphics

The output of this program is a prime candidate for visualization. In fact, some test ('make sure that turtles don't teleport') are hard to do other than by looking at the output. Start by making an ascii rendering of the ocean grid, as in figure 54.1.

It would be better to have some sort of animated output. However, not all programming languages generate visual output equally easily. There are very powerful video/graphics libraries in C++, but these are also hard to use. There is a simpler way out.

For simple output such as this program yields, you can make a simple low-budget animation. Every *terminal emulator* under the sun supports VT100 cursor control<sup>1</sup>: you can send certain magic output to your screen to control cursor positioning.

In each time step you would

1. Send the cursor to the home position, by this magic output:

```
// turtles/pacific.cpp
#include <cstdio>
/* ... */
// ESC [ i ; j H
printf( "%c[0;0H", (char)27 );
```

2. Display your grid as in figure 54.1;
3. Sleep for a fraction of a second; see section 25.1.4.

---

1. <https://vt100.net/docs/vt100-ug/chapter3.html>

---

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
0   o
1           o
2           |
3   o       o       o
4           x
5           o
6           o
7   o       x
8           o
9           o
0   o
1
2   o
3   o
4   o
5
6
7   o       o       o
8   |       o       o
9   o       o
0   o       o       o
1           o
2   o       o
3
4   o
5           x   x
6           |
7           x   x
8   |       o   o
9
0   x
1
2
3   o
4   o   |
5           x
Turtles: 55
Trash : 21
Ships : 12
```

Figure 54.1: Ascii art printout of a time step

## 54.4 Modern programming techniques

### 54.4.1 Object oriented programming

While there is only have one ocean, you should still make an *ocean* class, rather than having a global array object. All functions are then methods of the single object you create of that class.

### 54.4.2 Data structure

We lose very little generality by ignoring the depth of the ocean and the shape of coastlines, and model the ocean as a 2D grid.

If you write an indexing function `cell(i, j)` you can make your code largely independent of the actual data structure chosen. Argue why `vector<vector<int>>` is not the best storage.

1. What do you use instead?
2. When you have a working code, can you show by timing that your choice is indeed superior?

### 54.4.3 Cell types

Having ‘magic numbers’ in your code (0 =empty, 1 =turtle, et cetera) is not elegant. Make a `enum` or `enum class` (see section 24.10) so that you have names for what’s in your cells:

```
// turtles/pacific.cpp
cell(i, j) = occupy::turtle;
```

If you want to print out your ocean, it might be nice if you can directly `cout` your cells:

```
// turtles/pacific.cpp
    for (int i=0; i<iSize; ++i) {
        cout << i%10 << " ";
        for (int j=0; j<jSize; ++j) {
            cout << setw(hs) << cell(i, j);
        }
        cout << '\n';
    }
```

### 54.4.4 Ranging over the ocean

It is easy enough to write a loop as

```
for (int i=0; i<iSize; i++)
    for (int j=0; j<jSize; j++)
        ... cell(i, j) ...
```

However, it may not be a good idea to always sweep over your domain so orderly. Can you implement this:

```
for ( auto [i, j] : permuted_indices() ) {
    ... cell(i, j) ...
```

? See section 24.5 about *structured binding*.

Likewise, if you need to count how many pieces of trash there are around a turtle, can you get this code to work:

```
// turtles/pacific.cpp
int count_around( int ic, int jc, occupy typ ) const {
    int count=0;
    for ( auto [i,j] : neighbors(ic, jc) ) {
        if (cell(i,j) == typ)
            ++count;
    }
    return count;
};
```

#### 54.4.5 Random numbers

For the random movement of ships and turtles you need a random number generator. Do not use the old C generator, but the new `random` one; section 24.7.

Try to find a solution so that you use exactly one generator for all places where you need random numbers. Hint: make the generator `static` in your class.

### 54.5 Explorations

Instead of having the ships move randomly, can you give them a preferential direction to the closest garbage patch? Does this improve the health of the turtle population?

Can you account for the relative size of ships and turtles by having a ship occupy a  $2 \times 2$  block in your grid?

So far you have let trash stay in place. What if there are ocean currents? Can you make the trash ‘sticky’ so that trash particles start moving as a patch if they touch?

Turtles eat sardines. (No they don’t.) What happens to the sardine population if turtles die out? Can you come up with parameter values that correspond to a stable ecology or a de-stabilized one?

#### 54.5.1 Code efficiency

Investigate whether your implementation of the `enum` in section 54.4.3 has any effect on timing. Parse the fine print of section 24.10.

You may remark that ranging over a largely empty ocean can be pretty inefficient. You could contemplate keeping an ‘active list’ of where the turtles et cetera are located, and only looping over that. How would you implement that? Do you expect to see a difference in timing? Do you actually?

How is runtime affected by choosing a vector-of-vectors implementation for the ocean; see section 54.4.2.



# Chapter 55

## Graph algorithms

In this project you will explore some common graph algorithms, and their various possible implementations. The main theme here will be that the common textbook exposition of algorithms is not necessarily the best way to phrase them computationally.

As background knowledge for this project, you are encouraged to read HPC book [13], chapter 10; for an elementary tutorial on graphs, see HPC book [13], chapter 25.

### 55.1 Traditional algorithms

We first implement the ‘textbook’ formulations of two *Single Source Shortest Path (SSSP)* algorithms: on unweighted and then on weighted graphs. In the next section we will then consider formulations that are in terms of linear algebra.

In order to develop the implementations, we start with some necessary preliminaries,

#### 55.1.1 Code preliminaries

##### 55.1.1.1 Adjacency graph

We need a class `Dag` for a Directed Acyclic Graph (DAG):

```
// tree/dijkstra1.cpp
class Dag {
private:
    vector<vector<int>> dag;
public:
    // Make Dag of 'n' nodes, no edges for now.
    Dag( int n )
        : dag( vector<vector<int>>(n) ) { };
```

It’s probably a good idea to have a function

```
// tree/dijkstra1.cpp
const auto& neighbors( int i ) const { return dag.at(i); };
```

that, given a node, returns a list of the neighbors of that node.

**Exercise 55.1.** Finish the *Dag* class. In particular, add a method to generate example graphs:

- For testing the ‘circular’ graph is often useful: connect edges

$$0 \rightarrow 1 \rightarrow \dots \rightarrow N - 1 \rightarrow 0.$$

- It may also be a good idea to have a graph with random edges.

Write a method that gives a textual rendering of the graph.

#### 55.1.1.2 Node sets

The classic formulation of SSSP algorithms, such as the *Dijkstra shortest path algorithm* (see HPC book [13], section 10.1.3) uses sets of nodes that are gradually built up or depleted.

You could implement that as a vector:

```
vector< int > set_of_nodes(nnodes);
for ( int inode=0; inode<nnodes; inode++ )
    // mark inode as distance unknown:
    set_of_nodes.at(inode) = inf;
```

where you use some convention, such as negative distance, to indicate that a node has been removed from the set.

However, C++ has an actual `set` container with methods for adding an element, finding it, and removing it; see section 24.3.2. This makes for a more direct expression of our algorithms. In our case, we’d need a set of int/int or int/float pairs for the node number and edge weight, depending on the graph algorithm. (It is also possible to use a `map`, using an `int` as lookup key, and `int` or `float` as values.)

For the unweighted graph we only need a set of finished nodes, and we insert node 0 as our starting point:

```
// tree/queuelevel.cpp
using node_info = std::pair<unsigned,unsigned>;
std::set< node_info > distances;
distances.insert( {0,0} );
```

For Dijkstra’s algorithm we need both a set of finished nodes, and nodes that we are still working on. We again set the starting node, and we set the distance for all unprocessed nodes to infinity:

```
// tree/queuedijkstra.cpp
const unsigned inf = std::numeric_limits<unsigned>::max();
using node_info = std::pair<unsigned,unsigned>;
std::set< node_info > distances,to_be_done;

to_be_done.insert( {0,0} );
for (unsigned n=1; n<graph_size; ++n)
    to_be_done.insert( {n,inf} );
```

(Why do we need that second set here, while it was not necessary for the unweighted graph case?)

**Exercise 55.2.** Write a code fragment that tests if a node is in the *distances* set.

- You can of course write a loop for this. In that case know that iterating over a set gives you the key/value pairs. Use *structured bindings*; section 24.5.

- But it's better to use an 'algorithm', in the technical sense of 'algorithms built into the standard library'. In this case, `find`.
- ... except that with `find` you have to search for an exact key/value pair, and here you want to search: 'is this node in the *distances* set with whatever value'. Use the `find_if` algorithm; section 24.3.2.

### 55.1.2 Level set algorithm

We start with a simple algorithm: the SSSP algorithm in an unweighted graph; see HPC book [13], section 10.1.1 for details. Equivalently, we find level sets in the graph.

For unweighted graphs, the distance algorithm is fairly simple. Inductively:

- Assume that we have a set of nodes reachable in at most  $n$  steps,
- then their neighbors (that are not yet in this set) can be reached in  $n + 1$  steps.

The algorithm outline is

```
// tree/queuelevel.cpp
for (;;) {
    if (distances.size() == graph_size) break;
    /*
     * Loop over all nodes that are already done
     */
    for ( auto [node, level] : distances ) {
        /*
         * set neighbors of the node to have distance 'level + 1'
         */
        const auto& nbors = graph.neighbors(node);
        for ( auto n : nbors ) {
            /*
             * See if 'n' has a known distance,
             * if not, add to 'distances' with level+1
             */
            /*
             * ...
             */
            {
                cout << "node " << n << " level " << level+1 << '\n';
                distances.insert( {n, level+1} );
            }
        }
    }
}
```

**Exercise 55.3.** Finish the program that computes the SSSP algorithm and test it.

This code has an obvious inefficiency: for each level we iterate through all finished nodes, even if all their neighbors may already have been processed.

**Exercise 55.4.** Maintain a set of 'current level' nodes, and only investigate these to find the next level. Time the two variants on some large graphs.

### 55.1.3 Dijkstra's algorithm

In Dijkstra's algorithm we maintain both a set of nodes for which the shortest distance has been found, and one for which we are still determining the shortest distance. Note: a tentative shortest distance for a

node may be updated several times, since there may be multiple paths to that node. The ‘shortest’ path in terms of weights may not be the shortest in number of edges traversed!

The main loop now looks something like:

```
// tree/queuedijkstra.cpp
for (;;) {
    if (to_be_done.size() == 0) break;
    /*
     * Find the node with least distance
    */
    /* ... */
    cout << "min: " << nclose << " @ " << dclose << '\n';
    /*
     * Move that node to done,
    */
    to_be_done.erase(closest_node);
    distances.insert( *closest_node );
    /*
     * set neighbors of nclose to have that distance + 1
    */
    const auto& nbors = graph.neighbors(nclose);
    for ( auto n : nbors ) {
        // find 'n' in distances
        /*
        */
        {
            /*
             * if 'n' does not have known distance,
             * find where it occurs in 'to_be_done' and update
            */
            /*
            */
            to_be_done.erase( cfind );
            to_be_done.insert( {n, dclose+1} );
            /*
            */
        }
    }
}
```

(Note that we erase a record in the `to_be_done` set, and then re-insert the same key with a new value. We could have done a simple update if we had used a `map` instead of a `set`.)

The various places where you find nodes in the finished / unfinished sets are up to you to implement. You can use simple loops, or use `find_if` to find the elements matching the node numbers.

**Exercise 55.5.** Fill in the details of the above outline to realize Dijkstra’s algorithm.

## 55.2 Linear algebra formulation

In this part of the project you will explore how much you make graph algorithms look like linear algebra.

### 55.2.1 Code preliminaries

#### 55.2.1.1 Data structures

You need a matrix and a vector. The vector is easy:

```
// tree/graphmvpdijkstra.cpp
class Vector {
private:
    vector<vectorvalue> values;
public:
    Vector( int n )
        : values( n, infinite ) {};
}
```

For the matrix, use initially a dense matrix:

```
// tree/graphmvpdijkstra.cpp
class AdjacencyMatrix {
private:
    vector<vector<matrixvalue>> adjacency;
public:
    AdjacencyMatrix( int n )
        : adjacency( vector<vector<matrixvalue>>( n, vector<matrixvalue>( n, empty ) ) ) {
    };
}
```

but later we will optimize that.

**Remark** In general it's not a good idea to store a matrix as a vector-of-vectors (see section 10.8.1), but in this case we need to be able to return a matrix row, so it is convenient. Optionally you can explore the use of `mdspan`; section 10.8.3.1.

### 55.2.1.2 Matrix vector multiply

Let's write a routine

```
Vector AdjacencyMatrix::leftmultiply( const Vector& left );
```

This is the simplest solution, but not necessarily the most efficient one, as it creates a new vector object for each matrix-vector multiply.

As explained in the theory background (HPC book [13], section 10.2) graph algorithms can be formulated as matrix-vector multiplications with unusual add/multiply operations. Thus, the core of the multiplication routine could look like

```
// tree/graphmvp.cpp
for ( int row=0; row<n; ++row ) {
    for ( int col=0; col<n; ++col ) {
        result[col] = add( result[col], mult( left[row], adjacency[row][col] ) );
    }
}
```

### 55.2.2 Unweighted graphs

**Exercise 55.6.** Implement the `add` / `mult` routines to make the SSSP algorithm on unweighted graphs work.

### 55.2.3 Dijkstra's algorithm

As an example, consider the following adjacency matrix:

```
. 1 . . 5
. . 1 . .
. . . 1 .
. . . . 1
1 . . . .
```

The shortest distance  $0 \rightarrow 4$  is 4, but in the first step a larger distance of 5 is discovered. Your algorithm should show an output similar to this for the successive updates to the known shortest distances:

```
Input : 0 . . . .
step 0: 0 1 . . 5
step 1: 0 1 2 . 5
step 2: 0 1 2 3 5
step 3: 0 1 2 3 4
```

**Exercise 55.7.** Implement new versions of the `add` / `mult` routines to make the matrix-vector multiplication correspond to Dijkstra's algorithm for SSSP on weighted graphs.

### 55.2.4 Sparse matrices

The matrix data structure described above can be made more compact by storing only nonzero elements. Implement this.

## 55.3 Bonus: parallelism

Explore the OpenMP library to parallelize the various algorithms you have developed in this project.

Argue that a CRS sparse matrix representation is easy to parallelize. Realize this with OpenMP and measure speedup.

How hard is it to parallelize the implementation of a graphs as node sets, as you started this project? In this case the level sets are key to parallel processing. However, on any but very large graphs you may not see much speedup.

## 55.4 Further explorations

How elegant can you make your code through operator overloading?

Can you code the all-pairs shortest path algorithm?

Can you extend the SSSP algorithm to also generate the actual paths?

## 55.5 Tests and reporting

You now have two completely different implementations of some graph algorithms. Generate some large matrices and time the algorithms.

Discuss your findings, paying attention to amount of work performed and amount of memory needed.



# Chapter 56

## Congestion

This section contains a sequence of exercises that builds up to a simulation of car traffic.

### 56.1 Problem statement

Car traffic is determined by individual behaviors: drivers accelerating, braking, deciding to go one way or another. From this, emergent phenomena arise, most clearly of course traffic jams.

### 56.2 Code design

In this project we will explore the so-called *actor model*: the simulation consists of independent actors that react to each others' actions.

We need only two basic classes:

1. A *Car* class, where objects have a location and a speed, and they can react to the car in front of them. (For simplicity we assume that drivers only look at the car immediately ahead of them.)
2. a *Street* class, where each street is a container for a number of cars.

There are various ways we can run a simulation (with *threads*, driven by *interrupts*), but for simplicity we consider discrete time steps. This means that both the car and street class have a *progress* method that advanced the object by one time step.

#### 56.2.1 Cars

The main thing you can ask of a car, is to move by a time step. This means you need to have its location, speed, and direction. It makes sense for a car to know its own speed and location, but the direction derives from the street. Thus, the street could update the cars as:

```
// traffic/traffic_lib.cpp
for ( auto c : cars() ) {
    c->progress( unit_vector() );
}
```

and the car would take the direction as an input.

### 56.2.2 Street

Let's start with one-way streets. A street is then a collection of cars, but, until they start passing each other, they are actually ordered. A vector would be suitable for that. However, a car can move to a different street, or even be in two streets at once if it's on an intersection. For that reason, model a street as

```
class Street {  
private:  
    vector<shared_ptr<Car>> cars;
```

Once we get the simulation going, cars are going to react to each other's behavior, in particular the behavior of the car immediately in front of them. There are various ways you can model that.

1. For instance, the street can pass the information of one car to the next.
2. Possibly more elegant, each car has a pointer to the one before and after it.

### 56.2.3 Unit tests

At this level you can do a number of unit tests to ensure that your code behaves as intended.

1. Define a street, and test its unit vector.
2. Place a car in various locations, both on the street and beyond or the side of it. Write test methods for these cases, for instance
3. If you put a car on the street, its speed stays constant, and it leaves the street after a predictable amount of time:

```
// traffic/street.cpp  
REQUIRE_NO_THROW( main_street.insert_with_speed(speed) );  
float time=0.f;  
for ( int t=0; t<static_cast<int>(main_street.length()); ++t ) {  
    INFO( format( "at t={}, #cars={}", t, main_street.size() ) );  
    if ( main_street.empty() ) break;  
  
    if ( global_do_vis ) {  
        // animation:  
        main_street.display();  
        std::this_thread::sleep_for( seconds{2}/10. );  
        cout << '\n';  
        print( "%c[1A", (char)27); // line up again.  
    }  
  
    auto car = main_street.at(0);  
    REQUIRE( car->speed()==Catch::Approx(speed) );  
    REQUIRE_NO_THROW( main_street.progress() );  
}  
REQUIRE( main_street.empty() );
```

# Chapter 57

## DNA Sequencing

In this set of exercises you will write mechanisms for DNA sequencing.

### 57.1 Basic functions

Refer to section [24.5](#).

First we set up some basic mechanisms.

**Exercise 57.1.** There are four bases, A, C, G, T, and each has a complement: A  $\leftrightarrow$  T, C  $\leftrightarrow$  G. Implement this through a map, and write a function

```
char BaseComplement(char);
```

**Exercise 57.2.** Write code to read a *Fasta* file into a string. The first line, starting with >, is a comment; all other lines should be concatenated into a single string denoting the genome.

Read the virus genome in the file `lambda_virus.fa`.

Count the four bases in the genome two different ways. First use a map. Time how long this takes. Then do the same thing using an array of length four, and a conditional statement.

Bonus: try to come up with a faster way of counting. Use a vector of length 4, and find a way of computing the index directly from the letters A, C, G, T. Hint: research ASCII codes and possibly bit operations.

### 57.2 De novo shotgun assembly

One approach to generating a genome is to cut it to pieces, and algorithmically glue them back together. (It is much easier to sequence the bases of a short read than of a very long genome.)

If we assume that we have enough reads that each genome position is covered, we can look at the overlaps by the reads. One heuristic is then to find the Shortest Common Superset (SCS).

### 57.2.1 Overlap layout consensus

1. Make a graph where the reads are the vertices, and vertices are connected if they overlap; the amount of overlap is the edge weight.
2. The SCS is then a *Hamiltonian path* through this graph – this is already NP-complete.
3. Additionally, we optimize for maximum total overlap.  
⇒ Traveling Salesman Problem (TSP), NP-hard.

Rather than finding the optimal superset, we can use a greedy algorithm, where every time we find the read with maximal overlap.

Repeats are often a problem. Another is spurious subgraphs from sequencing errors.

### 57.2.2 De Bruijn graph assembly

### 57.3 ‘Read’ matching

A ‘read’ is a short fragment of DNA, that we want to match against a genome. In this section you will explore algorithms for this type of matching.

While here we mostly consider the context of genomics, such algorithms have other applications. For instance, searching for a word in a web page is essentially the same problem. Consequently, there is a considerable history of this topic.

#### 57.3.1 Naive matching

We first explore a naive matching algorithm: for each location in the genome, see if the read matches up.

```
ATACTGACCAAGAACGTGATTACTTCATGCAGCGTTACCAT  
ACCAAGAACGTG  
^ mismatch
```

```
ATACTGACCAAGAACGTGATTACTTCATGCAGCGTTACCAT  
ACCAAGAACGTG  
total match
```

**Exercise 57.3.** Code up the naive algorithm for matching a read. Test it on fake reads obtained by copying a substring from the genome. Use the genome in `phix.fa`.

Now read the *Fastq* file `ERR266411_1.first1000.fastq`. Fastq files contains groups of four lines: the second line in each group contains the reads. How many of these reads are matched to the genome?

Reads are not necessarily a perfect match; in fact, each fourth line in the fastq file gives an indication of the ‘quality’ of the corresponding read. How many matches do you get if you take a substring of the first 30 or so characters of each read?

#### 57.3.2 Boyer-Moore matching

The *Boyer-Moore* string matching algorithm [6] is much faster than naive matching, since it uses two clever tricks to weed out comparisons that would not give a match.

### Bad character rule

In naive matching, we determined match locations left-to-right, and then tried matching left-to-right. In Bowers-Moore (BM), we still find match locations left-to-right, but we do our matching right-to-left.

```
vvvv match location
antidisestablishmentarianism
    blis
        ^bad character
```

The mismatch is an ‘l’ in the pattern, which does not match a ‘d’ in the text. Since there is no ‘d’ in the pattern at all, we move the pattern completely past the mismatch:

```
vvvv match location
antidisestablishmentarianism
    blis
```

in fact, we move it further, to the first match on the first character of the pattern:

```
vvvv match location
antidisestablishmentarianism
    blis
        ^ first character match
```

The case where we have a mismatch, but the character in the text does appear in the pattern is a little trickier: we find the next occurrence of the mismatched character in the pattern, and use that to determine the shift distance.

```
shoobeedoobeeboobah
edoobeeboob
    ^ mismatch
    ^ other occurrence of 'd'
```

Note that this can be a considerably smaller shift than in the previous case.

```
v
shoobeedoobeeboobah
edoobeeboob
    ^ match the bad character 'd'
    ^ new location
```

**Exercise 57.4.** Discuss how efficient you expect this heuristic to be in the context of genomics versus text searching. (See above.)

### Good suffix rule

The ‘good suffix’ consists of the matched characters after the bad character. When moving the read, we try to keep the good suffix intact:

```
desistrust
```

```
listrest
    ^^good suffix

desistrust
listrest
    ^^next occurrence of suffix
```

## 57.4 Bloom filters

[https://en.wikipedia.org/wiki/Bloom\\_filters\\_in\\_bioinformatics](https://en.wikipedia.org/wiki/Bloom_filters_in_bioinformatics)

## Chapter 58

### Memory allocation

**This project is not yet ready**

<https://www.youtube.com/watch?v=R3cBbvIFqFk>

Monotonic allocator

- base and free pointer,
- always allocate from the free location
- only release when everything has been freed.

appropriate:

- video processing: release everything used for one frame
- event processing: release everything used for handling the event

Stack allocator



# **Chapter 59**

## **Ballistics calculations**

**THIS PROJECT IS NOT READY FOR PRIME TIME**

### **59.1      Introduction**

From <https://encyclopedia2.thefreedictionary.com/ballistics>

#### **Ballistics**

the science of the movement of artillery shells, bullets, mortar shells, aerial bombs, rocket artillery projectiles and missiles, harpoons, and so on. Ballistics is a technical military science based on a set of physics and mathematics disciplines. Interior ballistics is distinguished from exterior ballistics.

Interior ballistics is concerned with the movement of a projectile (or other body whose mechanical freedom is restricted by certain conditions) in the bore of a gun under the influence of powder gases as well as the rules governing other processes occurring in the bore or in the chamber of a powder rocket during firing. Interior ballistics views the firing as a complex process of rapid transformation of the powder's chemical energy into heat energy and then into the mechanical work of displacing the projectile, charge, and recoil parts of the gun. In interior ballistics the different periods that are distinguished in the firing are the preliminary period, which is from the start of the powder combustion until the projectile begins to move; the first (primary) period, which is from the start of projectile movement until the end of powder combustion; the second period, which is from the end of powder combustion until the moment that the projectile leaves the bore (the period of adiabatic expansion of the gases); and the period of the aftereffect of the powder gases on the projectile and barrel. The laws governing the processes related to this last period are dealt with in a special division of ballistics, known as intermediate ballistics. The end of the period of aftereffect on the projectile divides the phenomena studied by interior and exterior ballistics.

The main divisions of interior ballistics are "pyrostatics," "pyrodynamics," and ballistic gun design. Pyrostatics is the study of the laws of powder combustion and gas formation during the combustion of powder in a constant volume in which the effect of the chemical composition of the powder and its forms and dimensions on the laws of combustion and gas formation is determined. Pyrodynamics is concerned with the study of the processes and phenomena that take place in the

bore during firing and the determination of the relationships between the design features of the bore, the conditions of loading, and various physical-chemical and mechanical processes that occur during firing. On the basis of a consideration of these processes and also of the forces operating on the projectile and barrel, a system of equations is established that describes the firing process, including the basic equation of interior ballistics, which relates the magnitude of the burned part of the charge, the pressure of powder gases in the bore, the velocity of the projectile, and the length of the path it has traveled. The solution of this system and the discovery of the relationship between change in the pressure  $\rho$  of the powder gases, the velocity  $v$  of the projectile, and other parameters on path  $l$  of the projectile and the time it has moved along the bore is the first main (direct)

to solve this problem the analytic method, numerical integration methods (including those based on computers), and tabular methods are used. In view of the complexity of the firing process and insufficient study of particular factors, certain assumptions are made. The correction formulas of interior ballistics are of great practical significance; they make it possible to determine the change in muzzle velocity of the projectile and maximum pressure in the bore when there are changes made in the loading conditions.

Ballistic gun design is the second main (correlative) problem of interior ballistics. By it are determined the design specifications of the bore and the loading conditions under which a projectile of given caliber and mass will obtain an assigned (muzzle) velocity in flight. The curves of change in the pressure of the gases in the bore and of the velocity of the projectile along the length of the barrel and in time are calculated for the variation of the barrel selected during designing. These curves are the initial data in designing the artillery system as a whole and the ammunition for it. Internal ballistics also includes the study of the firing process in the rifle, in cases when special and combined charges are used, in systems with conical barrels, and in systems in which gases are exhausted during powder combustion (high-low pressure guns and recoilless guns, infantry mortars). Another important division is the interior ballistics of powder rockets, which has developed into a special science. The main divisions of the interior ballistics of powder rockets are pyrostatics of a semiclosed space, which consider the laws of powder combustion at comparatively low and constant pressure; the solution of the basic problem of the interior ballistics of powder rockets, which is to determine (under set loading conditions) the rules of variation in pressure of the powder gases in the chamber with regard to time and to determine the rules of variation in thrust necessary to ensure the required rocket velocity; the ballistic design of powder rockets, which involves determining the energy-producing characteristics of the powder, the weight and form of the charge, and the design parameters of the nozzle which ensure, with an assigned weight of the rocket's warhead, the necessary thrust force during its operation.

Exterior ballistics is concerned with the study of the movement of unguided projectiles (mortar shells, bullets, and so on) after they leave the bore (or launcher) and the factors that affect this movement. It includes basically the study of all the elements of motion of the projectile and of the forces that act upon it in flight (the force of air resistance, the force of gravity, reactive force, the force arising in the aftereffect period, and so on); the study of the movement of the center of mass of

the projectile for the purpose of calculating its trajectory (see Figure 2) when there are set initial and external conditions (the basic problem of exterior ballistics); and the determination of the flight stability and dispersion of projectiles. Two important divisions of exterior ballistics are the theory of corrections, which develops methods of evaluating the influence of the factors that determine the projectile's flight on the nature of its trajectory, and the technique for drawing up firing tables and of finding the optimal exterior ballistics variation in the designing of artillery systems. The theoretical solution of the problems of projectile movement and of the problems of the theory of corrections amounts to making up equations for the projectile's movement, simplifying these equations, and seeking methods of solving them. This has been made significantly easier and faster with the appearance of computers. In order to determine the initial conditions—that is, initial velocity and angle of departure, shape and mass of the projectile—which are necessary to obtain a given trajectory, special tables are used in exterior ballistics. The working out of the technique for drawing up a firing table involves determining the optimal combination of theoretical and experimental research that makes it possible to obtain firing tables of the required accuracy with the minimal expenditure of time. The methods of exterior ballistics are also used in the study of the laws of movement of spacecraft (during their movement without the influence of controlling forces and moments). With the appearance of guided missiles, exterior ballistics played a major part in the formation and development of the theory of flight and became a particular instance of this theory.

The appearance of ballistics as a science dates to the 16th century. The first works on ballistics are the books by the Italian N. Tartaglia, *A New Science* (1537) and *Questions*

and *Discoveries Relating to Artillery Fire* (1546). In the 17th century the fundamental principles of exterior ballistics were established by Galileo, who developed the parabolic theory of projectile movement, by the Italian E. Torricelli, and by the Frenchman M. Mersenne, who proposed that the science of the movement of projectiles be called ballistics (1644). I. Newton made the first investigations of the movement of a projectile, taking air resistance into consideration (*Mathematical Principles of Natural Philosophy*, 1687). During the 17th and 18th centuries the movement of projectiles was studied by the Dutchman C. Huygens, the Frenchman P. Varignon, the Englishman B. Robins, the Swiss D. Bernoulli, the Russian scientist L. Eiler, and others. The experimental and theoretical foundations of interior ballistics were laid in the 18th century in works of Robins, C. Hutton, Bernoulli, and others. In the 19th century the laws of air resistance were established (the laws of N. V. Maievskii and N. A. Zabudskii, Havre's law, and A. F. Siacci's law).

The numerical analysis of ballistics calculations on the ENIAC are described in [23].

### 59.1.1 Physics

These are the governing equations:

$$\begin{aligned} x'' &= -E(x' - w_x) + 2\Omega \cos L \sin \alpha y' \\ y'' &= -Ey' - g - 2\Omega \cos L \sin \alpha x' \\ z'' &= -E(z' - w_z) + 2\Omega \sin L x' + 2\Omega \cos L \cos \alpha y' \end{aligned} \tag{59.1}$$

where

- $x, y, z$  are the quantities of interest: distance, altitude, sideways displacement;
- $w_x, w_z$  are wind speed;
- $E$  is a complicated function of  $y$ , involving air density and speed of sound;
- $\alpha$  is the azimuth, that is, angle of firing;
- All other quantities are needed for physical realism, but will be set  $\equiv 1$  in this coding exercise.

### 59.1.2 Numerical analysis

This uses an Euler-MacLaurin scheme of third order:

$$f_1 - f_0 = \frac{1}{2}(f'_0 + f'_1)h + \frac{1}{12}(f'_0 - f'_1)h^2 + O(h^5) \quad (59.2)$$

which works out to

$$\begin{aligned} \bar{x}'_1 &= x'_0 + x''_0 \Delta t \\ x_1 &= x_0 + x'_0 \Delta t \\ x'_1 &= x'_0 + (x''_0 + \bar{x}''_1) \frac{\Delta t}{2} \\ x_1 &= x_0(x'_0 + \bar{x}'_1) \frac{\Delta t}{2} + (x''_0 - \bar{x}''_1) \frac{\Delta t^2}{12} \end{aligned} \quad (59.3)$$

## Chapter 60

### Cryptography

#### 60.1 The basics

While floating point numbers can span a rather large range – up to  $10^{300}$  or so for double precision – integers have a much smaller one: up to about  $10^9$ . That is not enough to do cryptographic applications, which deal in much larger numbers. (Why can't we use floating point numbers?)

So the first step is to write classes *Integer* and *Fraction* that have no such limitations. Use operator overloading to make simple expressions work:

```
Integer big=2000000000; //two billion  
big *= 1000000; bigger = big+1;  
Integer one = bigger % big;
```

**Exercise 60.1.** Code Farey sequences.

#### 60.2 Cryptography

[https://simple.wikipedia.org/wiki/RSA\\_algorithm](https://simple.wikipedia.org/wiki/RSA_algorithm)

[https://simple.wikipedia.org/wiki/Exponentiation\\_by\\_squaring](https://simple.wikipedia.org/wiki/Exponentiation_by_squaring)

#### 60.3 Blockchain

Implement a blockchain algorithm.



# Chapter 61

## Climate change

*The climate has changed and it is always changing.*

Raj Shah, White House Principal Deputy Press Secretary

The statement that climate always changes is far from a rigorous scientific claim. We can attach a meaning to it, if we interpret it as a statement about the statistical behavior of the climate, in this case as measured by average global temperature. In this project you will work with real temperature data, and do some simple analysis on it. (The inspiration for this project came from [24].)

Ideally, we would use data sets from various measuring stations around the world. Fortran is then a great language because of its array operations (see chapter 39): you can process all independent measurements in a single line. To keep things simple we will use a single data file here that contains data for each month in a time period 1880–2018. We will then use the individual months as ‘pretend’ independent measurements.

### 61.1 Reading the data

In the repository you find two text files

`GLB.Ts+dSST.txt`      `GLB.Ts.txt`

that contain temperature deviations from the 1951–1980 average. Deviations are given for each month of each year 1880–2018. These data files and more can be found at <https://data.giss.nasa.gov/gistemp/>.

**Exercise 61.1.** Start by making a listing of the available years, and an array `monthly_deviation` of size  $12 \times \text{nyears}$ , where `nyears` is the number of full years in the file. Use formats and array notation.

The text files contain lines that do not concern you. Do you filter them out in your program, or are you using a shell script? Hint: a judicious use of `grep` will make the Fortran code much easier.

### 61.2 Statistical hypothesis

We assume that Mr Shah was really saying that climate has a ‘stationary distribution’, meaning that highs and lows have a probability distribution that is independent of time. This means that in  $n$  data points,

each point has a chance of  $1/n$  to be a record high. Since over  $n + 1$  years each year has a chance of  $1/(n + 1)$ , the  $n + 1$ st year has a chance  $1/(n + 1)$  of being a record.

We conclude that, as a function of  $n$ , the chance of a record high (or low, but let's stick with highs) goes down as  $1/n$ , and that the gap between successive highs is approximately a linear function of the year<sup>1</sup>.

This is something we can test.

**Exercise 61.2.** Make an array `previous_record` of the same shape as `monthly_deviation`. This array records (for each month, which, remember, we treat like independent measurements) whether that year was a record, or, if not, when the previous record occurred:

$$\text{PrevRec}(m, y) = \begin{cases} y & \text{if } \text{MonDev}(m, y) = \max_{m'} (\text{MonDev}(m', y)) \\ y' & \text{if } \text{MonDev}(m, y) < \text{MonDev}(m, y') \\ & \text{and } \text{MonDev}(m, y') = \max_{m'' < m'} (\text{MonDev}(m'', y)) \end{cases}$$

Again, use array notation. This is also a great place to use the `Where` clause.

**Exercise 61.3.** Now take each month, and find the gaps between records. This gives you two arrays: `gapyears` for the years where a gap between record highs starts, and `gapsizes` for the length of that gap.

This function, since it is applied individually to each month, uses no array notation.

The hypothesis is now that the `gapsizes` are a linear function of the year, for instance measured as distance from the starting year. Of course they are not exactly a linear function, but maybe we can fit a linear function through it by *linear regression*.

**Exercise 61.4.** Copy the code from <http://www.aip.de/groups/soe/local/numres/bookfpdf/f15-2.pdf> and adapt for our purposes: find the best fit for the slope and intercept for a linear function describing the gaps between records.

You'll find that the gaps are decidedly not linearly increasing. So is this negative result the end of the story, or can we do more?

**Exercise 61.5.** Can you turn this exercise into a test of global warming? Can you interpret the deviations as the sum of a yearly increase in temperature plus a stationary distribution, rather than a stationary distribution by itself?

---

1. Technically, we are dealing with a uniform distribution of temperatures, which makes the maxima and minima have a beta-distribution.

## Chapter 62

### Desk Calculator Interpreter

In this set of exercises you will write a ‘desk calculator’: a small interactive calculator that combines numerical and symbolic calculation.

These exercises mostly use the material of chapters 36, 41, 35.

#### 62.1 Named variables

We start out by working with ‘named variables’: the *namedvar* type associates a string with a variable:

```
// structf/varhandling.F90
type namedvar
    character(len=20) :: expression = ""
    integer :: value
end type namedvar
```

A named variable has a value, and a string field that is the expression that generated the variable. When you create the variable, the expression can be anything.

```
// structf/varhandling.F90
type(namedvar) :: x,y,z,a
x = namedvar("x",1 )
y = namedvar("yvar",2 )
```

Next we are going to do calculations with these type objects. For instance, adding two objects

- adds their values, and
- concatenates their *expression* fields, giving the expression corresponding to the sum value.

Your first assignment is to write *varadd* and *varmult* functions that get the following program working with the indicated output. This uses string manipulation from sections 35.3 and 41.5.

**Exercise 62.1.** The following main program should give the corresponding output:

|                                                                                                                                                                                                                                                                                                                                |   |   |      |   |              |   |                      |   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---|------|---|--------------|---|----------------------|---|
| <b>Code :</b><br><pre>// structf/varhandling.F90 print *,x print *,y z = varadd(x,y) print *,z a = varmult(x,z) print *,a</pre>                                                                                                                                                                                                |   |   |      |   |              |   |                      |   |
| <b>Output :</b><br><b>[structf] varhandling:</b> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;">x</td> <td style="width: 30%;">1</td> </tr> <tr> <td>yvar</td> <td>2</td> </tr> <tr> <td>(x) + (yvar)</td> <td>3</td> </tr> <tr> <td>(x) * ((x) + (yvar))</td> <td>3</td> </tr> </table> | x | 1 | yvar | 2 | (x) + (yvar) | 3 | (x) * ((x) + (yvar)) | 3 |
| x                                                                                                                                                                                                                                                                                                                              | 1 |   |      |   |              |   |                      |   |
| yvar                                                                                                                                                                                                                                                                                                                           | 2 |   |      |   |              |   |                      |   |
| (x) + (yvar)                                                                                                                                                                                                                                                                                                                   | 3 |   |      |   |              |   |                      |   |
| (x) * ((x) + (yvar))                                                                                                                                                                                                                                                                                                           | 3 |   |      |   |              |   |                      |   |

(To be clear: the two routines need to do both numeric and string ‘addition’ and ‘multiplication’.)

You can base this off the file `namedvar.F90` in the repository

## 62.2 First modularization

Let’s organize the code so far by introducing modules; see chapter 37.

**Exercise 62.2.** Create a module (suggested name: `VarHandling`) and move the `namedvar` type definition and the routines `varadd`, `varmult` into it.

**Exercise 62.3.** Also create a module (suggested name: `InputHandling`) that contains the routines `islower`, `isdigit` from the character exercises in chapter 35. You will also need an `isop` routine to recognize arithmetic operations.

## 62.3 Event loop and stack

In our quest to write an interpreter, we will write an ‘event loop’: a loop that continually accepts single character inputs, and processes them. An input of "0" will mean termination of the process.

**Exercise 62.4.** Write a loop that accepts character input, and only prints out what kind of character was encountered: a lowercase character, a digit, or a character denoting an arithmetic operation `+*/`.

|                                                                                                                                                                                                                                                                                                                       |                   |              |                  |              |                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|--------------|------------------|--------------|------------------|
| <b>Code :</b><br><pre>// structf/interchar.F90 do   read *,input   if (input .eq. '0') then     exit   else if ( isdigit(input) ) then</pre>                                                                                                                                                                          |                   |              |                  |              |                  |
| <b>Output :</b><br><b>[structf] interchar:</b> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;">Inputs: 4 x 3 + 0</td> </tr> <tr> <td>4 is a digit</td> </tr> <tr> <td>x is a lowercase</td> </tr> <tr> <td>3 is a digit</td> </tr> <tr> <td>+ is an operator</td> </tr> </table> | Inputs: 4 x 3 + 0 | 4 is a digit | x is a lowercase | 3 is a digit | + is an operator |
| Inputs: 4 x 3 + 0                                                                                                                                                                                                                                                                                                     |                   |              |                  |              |                  |
| 4 is a digit                                                                                                                                                                                                                                                                                                          |                   |              |                  |              |                  |
| x is a lowercase                                                                                                                                                                                                                                                                                                      |                   |              |                  |              |                  |
| 3 is a digit                                                                                                                                                                                                                                                                                                          |                   |              |                  |              |                  |
| + is an operator                                                                                                                                                                                                                                                                                                      |                   |              |                  |              |                  |

Use the `InputHandling` module introduced above.

### 62.3.1 Stack

Next, we are going to store values in `namedvar` types on a stack. A *stack* is a data structure where new elements go on the top, so we need to indicate with a *stack pointer* that top element. Equivalently, the stack pointer indicates how many elements there already are:

```
// structf/interpret.F90
type(namedvar), dimension(10) :: stack
integer :: stackpointer=0
```

Since we are using modules, let's keep the stack out of the main program and put it in the appropriate module.

**Exercise 62.5.** Add the `stack` variable and the stack pointer to the `VarHandling` module.

Since Fortran uses 1-based indexing, a starting value of zero is correct. For C/C++ it would have been `-1`.

Next we will start implementing stack operations, such as putting `namedvar` objects on the stack.

### 62.3.2 Stack operations

We extend the above event loop, which was only recognizing the input characters, by actually incorporating actions. That is, we repeatedly

1. read a character from the input;
2. 0 causes the event loop to exit; otherwise:
3. if it is a digit, create a new `namedvar` entry on the top of the stack, with that value both numerically as the `value` field, and as string in the `expression` field.

You may be tempted to write the following in the main program:

```
// structf/interpret.F90
if ( isdigit(input) ) then
    stackpointer = stackpointer + 1
    read( input,'( i1 )' ) stack(stackpointer)%value
    stack(stackpointer)%expression = trim(input)
```

(You have already coded `isdigit` in exercise 35.1.) but a cleaner design uses a function call to a method in the `VarHandling` module:

```
// structf/internum.F90
else if ( isdigit(input) ) then
    call stack_push(input)                                // structf/interpretm.F90
   subroutine stack_push(input)
   implicit none
   character,intent(in) :: input
```

Note that the `stack_push` routine does not have the stack or stack pointer as arguments: since they are all in the same module, they are accessible as *global variable*.

Finally,

4. if it is a letter indicating an operation `+, -, ×, /`,

- (a) take the two top entries from the stack, lowering the stack pointer;
- (b) apply that operation to the operands; and
- (c) push the result onto the stack.

The auxiliary function `stack_display` is a little tricky, so you get that here. This uses string formatting (section 41.3) and implied do loops (section 32.3): Also, note that the `stack` array and the `stackpointer` act like global variables.

```
// structf/internum.F90
subroutine stack_display()
    implicit none
    ! local variables
    integer :: istck

    if (stackpointer.eq.0) return
    print '( 10( a,a, a,i0,"; " ) )', ( &
        " expr=",trim(stack(istck)%expression), &
        " val=",stack(istck)%value, &
        istck=1,stackpointer )

end subroutine stack_display
```

Let's add the various options to the event loop.

**Exercise 62.6.** Make your event loop accept digits, creating a new entry:

**Code:**

```
// structf/internum.F90
else if ( isdigit(input) ) then
    call stack_push(input)
```

**Output:**

[structf] internum:

```
Inputs: 4 5 6 0
expr=4 val=4;
expr=4 val=4;  expr=5 val=5;
expr=4 val=4;  expr=5 val=5;  expr=6 val=6;
```

Next we integrate the operations: if the `input` character corresponds to an arithmetic operator, we call `stack_op` with that character. That routine in turn calls the appropriate operation depending on what the character was.

**Exercise 62.7.** Add a clause to your event loop to handle characters that stand for arithmetic operations:

**Code:**

```
// structf/internum.F90
else if ( isop(input) ) then
  call stack_op(input)
```

**Output:**

```
[structf] internumop:
Inputs: 4 5 6 + + 0
expr=4 val=4;
expr=4 val=4; expr=5 val=5;
expr=4 val=4; expr=5 val=5; expr=6 val=6;
expr=4 val=4; expr=(5)+(6) val=11;
expr=(4)+((5)+(6)) val=15;
```

### 62.3.3 Item duplication

Finally, we may want to use a stack entry more than once, so we need the functionality of duplicating a stack entry.

For this we need to be able to refer to a stack entry, so we add a single character label field: the *namedvar* type now stores

1. a single character id,
2. an integer value, and
3. its generating expression as string.

```
// structf/vartype.F90
type namedvar
  character :: id
  character(len=20) :: expression
  integer :: value
end type namedvar
```

**Exercise 62.8.** Add the *id* field to the *namedvar*, and make sure your program still compiles and runs.

The event loop is now extended with an extra step. If the input character is a lowercase letter, it is used as the *id* of a *namedvar* as follows.

- If there is already a stack entry with that *id*, it is duplicated on top of the stack;
- otherwise, the *id* of the stack top entry is set to this character.

Here is the relevant bit of the new *stack\_print* function:

```
// structf/interprets.F90
print '( 10( a,a1, a,a, a,i0,"; ") )', ( &
  "id:",stack(istck)%id, &
  " expr=",trim(stack(istck)%expression), &
  " val=",stack(istck)%value, &
  istck=1,top )
```

**Exercise 62.9.** Write the missing function and its clause in the event loop:

**Code:**

```
// structf/interpret.F90
stacksearch = find_on_stack(stack, stackpointer, input)
if ( stacksearch>=1 ) then
    stackpointer = stackpointer+1
    stack(stackpointer) = stack(stacksearch)
```

**Output:**

[structf] **stackfind:**

```
Inputs: 1 x 2 y x y + z 0
id:. expr=1 val=1;
id:x expr=1 val=1;
id:x expr=1 val=1; id:. expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2; id:x expr=1 val=1;
id:x expr=1 val=1; id:y expr=2 val=2; id:x expr=1 val=1; id:y expr=2
    ↗val=2;
id:x expr=1 val=1; id:y expr=2 val=2; id:. expr=(1)+(2) val=3;
id:x expr=1 val=1; id:y expr=2 val=2; id:z expr=(1)+(2) val=3;
```

(What is in the **else** part of this conditional?)

## 62.4 Modularizing

With the modules and the functions you have developed so far, you have a very clean main program:

```
// structf/intermod.F90
do
    call stack_display()
    read *,input
    if (input .eq. '0') exit
    if ( isdigit(input) ) then
        call stack_push(input)
    else if ( isop(input) ) then
        call stack_op(input)
    else if ( islower(input) ) then
        call stack_name(input)
    end if
end do
```

You see that by moving the stack into the module, neither the stack variable nor the stack pointer are visible in the main program anymore.

But there is an important limitation to this design: there is exactly one stack, declared as a sort of global variable, accessible through a module.

Whether having global data is good practice is another matter. In this case it's defensible: in a calculator app there will be exactly one stack.

## 62.5 Object orientation

But maybe we do sometimes need more than one stack. Let's bundle up the stack array and the stack pointer in a new type:

```
// structf/interclass.F90
type stackstruct
    type(namedvar), dimension(10) :: data
    integer :: top=0
contains
    procedure,public :: display, find_id, name, op, push
end type stackstruct
```

**Exercise 62.10.** Change the event loop so that it calls methods of the `stackstruct` type, rather than functions that take the stack as input.

For instance, the `push` function is called as:

```
// structf/interclass.F90
if ( isdigit(input) ) then
    call thestack%push(input)
```

### 62.5.1 Operator overloading

The `varadd` and similar arithmetic routines use a function call for what we would like to write as an arithmetic operation.

**Exercise 62.11.** Use operator overloading in the `varop` function:

```
// structf/interpretov.F90
if (op=="+") then
    varop = op1 + op2
```

et cetera.



**PART V**

**ADVANCED TOPICS**



# Chapter 63

## External libraries

If you have a C++ compiler, you can write as much software as you want, by yourself. However, some things that you may need for your work have already been written by someone else, and with a little luck that other programmer has published the code as a library.

This chapter teaches you how to install and use libraries, and will discuss a couple of libraries that are useful in the context of scientific computing.

### 63.1 What are software libraries?

In this chapter you will learn about the use of *software libraries*: software that is written not as a standalone package, but in such a way that you can access its functionality in your own program.

Software libraries can be enormous, as is the case for scientific libraries, which are often multi-person multi-year projects. On the other hand, many of them are fairly simple utilities written by a single programmer. In the latter case you may have to worry about future support of that software.

#### 63.1.1 Using an external library

Using a software library typically means that

- your program has a line

```
#include "fancylib.h"
```

- and you compile and link as:

```
icpc -c yourprogram.cpp -I/usr/include/fancylib  
icpc -o yourprogram yourprogram.o \  
-L/usr/lib/fancylib -lfancy
```

You will see specific examples below.

**Remark** There are so-called header-only libraries, meaning that they consist only of files to be included, and have no library files. In that case, of course, nothing has to be added to the link line.

If you are now worried about having to do a lot of typing every time you compile,

- if you use an IDE, you can typically add the library in the options, once and for all; or
- you can use *Make* or *CMake* for automating the build process of your program.

These include and library paths can get quite long – you certainly don't want to type them by hand. You could write a little shell script, but it's better to learn Make (see Tutorials book [11], section 3) or CMake (see Tutorials book [11], section 4).

Use ‘package config’ to find the library,  
then use variables with include and library paths/names

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( OPTS REQUIRED cxxopts )
target_include_directories(
    ${PROGRAM_NAME} PUBLIC
    ${OPTS_INCLUDE_DIRS}
)
```

### 63.1.2 Obtaining and installing an external library

Sometimes a software library is available through a *package manager* (macports and homebrew on Apple; yum, dpkg, rpm, … on Linux), but we are going to do it the old-fashioned way: downloading and installing it ourselves.

A popular location for finding downloadable software is [github.com](https://github.com), a hosting site for *Git* repositories. You can then choose whether to

- download everything in one file, typically with `.tgz` or `.tar.gz` extension, which you unpack with `tar`; or
- clone the repository, which gives you access to unreleased development software, or at least makes updating your copy easier. This usually gives you a directory with a name such as `fancylib-1.0.0` containing the source and documentation of the library, but not any binaries or machine-specific files.

Either way, from here on we assume that you have a directory containing the downloaded package.

There are two main types of installation:

- based on *GNU autotools*, which you recognize by the presence of a `configure` program; you then install your software in the following three-step process:

```
configure ## lots of options
make
make install
```

or

- based on *CMake*, which you recognize by the presence of `CMakeLists.txt` file:

```
cmake ## lots of options
make
make install
```

**Remark** Both `configure` and CMake will by default try to install your software in a system location such as `/usr/local/lib`. If you don't have permission to write there, you can supply an alternate location:

```
configure --prefix=/home/myname/mylibs
cmake -D CMAKE_INSTALL_PREFIX=/home/myname/mylibs
```

### 63.1.2.1 CMake installation

The easiest way to install a package using `cmake` is to create a build directory, next to the source directory. The `cmake` command is issued from this directory, and it references the source directory:

```
mkdir build
cd build
cmake ../fancylib-1.0.0
make
make install
```

Some people put the build directory inside the source directory, but that is bad practice.

Apart from specifying the source location, you can give more options to `cmake`. The most common are

- specifying an install location, for instance because you don't have *superuser* privileges on that machine; or
- specifying the compiler, because CMake will by default use the `gcc` compilers, but you may want the Intel compiler.

```
CC=icc CXX=icpc \
cmake \
-D CMAKE_INSTALL_PREFIX:PATH=${HOME}/mylibs/fancy \
../fancylib-1.0.0
```

For more on `cmake`, see Tutorials book [11], section 4.

## 63.2 Options processing: `cxxopts`

Suppose you have a program that does something with a large array. How are you specifying the size of the array? You can hardcode it:

```
vector<double> simulation(10000000);
```

which requires you to recompile the program if the amount of data ever changes. You can read it in from the command line:

```
size_t ndata;
cin >> ndata;
vector<double> simulation(ndata);
```

which requires you to enter this every time you run the program, and this makes it inconvenient to incorporate in a shell script.

Instead, you may want to pass this size as a commandline argument:

```
./myprogram --ndata 10000000
```

which is easier to retrieve from the shell *command history*, or put in a shell script.

Commandline arguments enter your code through the optional `argc`, `argv` arguments of the `main`. We start with a brief recap of how you could parse these yourself, but then go into the `cxxopts` library that makes this easy.

### 63.2.1 Traditional commandline parsing

Commandline options are available to the program through the (optional) `argv` and `argc` options of the `main` program. The former is an array of strings, with the second the length of that array, noting that index 0 is the name of the program, so `argc` is always at least 1.

```
int main( int argc, char **argv ) { /* program */ };
```

For simple cases it would be feasible to parse such options yourself:

| Code:                                                                                                                                                                                                                                                                            | Output:                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>// args/argcv.cpp cout &lt;&lt; "Program name: "     &lt;&lt; argv[0] &lt;&lt; '\n'; for (int iarg=1; iarg&lt;argc; ++iarg)     cout &lt;&lt; "arg: " &lt;&lt; iarg         &lt;&lt; argv[iarg] &lt;&lt; " =&gt; "         &lt;&lt; atoi( argv[iarg] ) &lt;&lt; '\n';</pre> | <pre>[args] argcv: ./argcv 5 12 Program name: ./argcv arg 1: 5 =&gt; 5 arg 2: 12 =&gt; 12 ./argcv abc 3.14 foo Program name: ./argcv arg 1: abc =&gt; 0 arg 2: 3.14 =&gt; 3 arg 3: foo =&gt; 0</pre> |

but this gets tedious quickly, and is difficult to make robust. Therefore, we will now look at a library that makes such options handling relatively easy.

### 63.2.2 The `cxxopts` library

We will now discuss the `cxxopts` ‘commandline argument parser’. It lets you construct an options object that has the information on all possible flags, any default values, and what types they can take.

First of all you need to include the header

```
#include "cxxopts.hpp"
```

We will now discuss the workings of this library.

#### 63.2.2.1 Simple example

The following shows how to declare two options, one for an integer value, and one getting a help message. The code tests for the help option: if it is present, it display the help message and exits; otherwise it tests for the numeric option.

```
Code:

// args/cxxn.cpp
cxxopts::Options options
    ("cxxopts",
     "Commandline options demo");
options.add_options()
    ("n,ntimes", "number of times",
     cxxopts::value<int>()
     ->default_value("37")
    )
    ("h,help", "usage information")
;
auto result = options.parse(argc, argv);
if (result.count("help")>0) {
    cout << options.help() << '\n';
    return 0;
}
auto number_of_times = result["ntimes"].as<
    int>();
cout << "number of times: "
    << number_of_times << '\n';
```

```
Output:
[args] cxxn:

./cxxn -h
Commandline options demo
Usage:
    cxxopts [OPTION...]

-n, --ntimes arg number
    ↪of times (default: 37)
-h, --help           usage
    ↪information

./cxxn -n 5 foo
number of times: 5
```

### 63.2.2.2 Discussion

Here is a more detailed listing of the actions that are needed or possible to use `cxxoptions`.

Declare an options object:

```
cxxopts::Options options("programname", "Program description");
```

Add options:

```
// args/cxxopts.cpp
// define '-n 567' option:
options.add_options()
    ("n,ntimes", "number of times",
     cxxopts::value<int>()
     ->default_value("37")
    )
    ;
/* ... */
// read out '-n' option and use:
auto number_of_times = result["ntimes"].as<int>();
cout << "Using number of times: " << number_of_times << '\n';
```

Add array options

```
// args/cxxopts.cpp
//define '-a 1,2,5,7' option:
options.add_options()
    ("a,array", "array of values",
     cxxopts::value< vector<int> >() ->default_value("1,2,3")
    )
    ;
```

```
/* ... */
auto array = result["array"].as<vector<int>>();
cout << "Array: " ;
for ( auto a : array ) cout << a << ", ";
cout << '\n';
```

Add positional arguments:

```
// args/cxxopts.cpp
// define 'positional argument' option:
options.add_options()
    ("keyword","whatever keyword",
     cxxopts::value<string>())
;
options.parse_positional({ "keyword" });
/* ... */

// read out keyword option and use:
auto keyword = result["keyword"].as<string>();
cout << "Found keyword: " << keyword << '\n';
```

Parse the options:

```
auto result = options.parse(argc, argv);
```

Get help flag first:

```
// args/cxxopts.cpp
options.add_options()
    ("h,help","usage information")
;
/* ... */

auto result = options.parse(argc, argv);
if (result.count("help")>0) {
    cout << options.help() << '\n';
    return 0;
}
```

Options can be specified the usual ways:

```
myprogram -n 10
myprogram --nsize 100
myprogram --nsize=1000
myprogram --array 12,13,14,15
```

**Exercise 63.1.** Incorporate this package into primality testing: exercise 45.26.

Options parsing can throw a `cxxopts::exceptions::option_has_no_value` exception.

### 63.2.3 Install and usage

The library can be downloaded or cloned at <https://github.com/jarro2783/cxxopts>.

Installation with CMake is straightforward.

The `cxxopts` package can be discovered in `CMake` through `pkgconfig`:

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( CXXOPTS REQUIRED cxxopts )
target_include_directories( ${PROJECT_NAME} PUBLIC ${CXXOPTS_INCLUDE_DIRS} )
```

Alternatively, you can use the compile option:

```
-I/path/to//cxxopts/installdir/include
```

### 63.3 Linear algebra libraries

Linear algebra operations feature in many scientific applications. For instance, many PDE problems require solving linear systems of equations, or eigenvalue problems. In recent year, ML has given us another application of matrix operations.

So it is not surprising that there are many software libraries that implement common linear algebra operations. These not only save you a lot of typing, but they typically also have higher performance and better numerical properties than your own implementation would have.

Some of the common libraries are:

- *BLAS*. This contains fairly simple operations on matrices and vectors. Note that ‘Blas’ is more an interface specification than an actual library. The ‘reference implementation’ is written in Fortran, but higher quality implementations are offered in the open source *BLIS* package, and proprietary packages such as Intel’s *MKL*. The fact that the reference is in Fortran means that calling BLAS operations from C++ can feel needlessly complicated.
- *Lapack*. Building on the BLAS operations, this offers the solution of dense linear systems, and eigenvalue solvers, Again, the reference implementation is written in Fortran.
- *Eigen*. This is a completely C++-based library giving elegant access to BLAS and Lapack functionality, as well as sparse matrix operations. For serious scientific computing its drawback is the poor support for parallelism.
- Finally, there are several libraries designed to be executed in parallel: *PETSc*, *Trilinos*, *Hypre*, *Mumps*. These are not necessarily written in C++, and may therefore feel a little non-idiomatic. However, their functionality is valuable enough that you will gladly put up with that.



## Chapter 64

### Programming strategies

#### 64.1 A philosophy of programming

Yes, your code will be executed by the computer, but:

- You need to be able to understand your code a month or year from now.
- Someone else may need to understand your code.
- ⇒ make your code readable, not just efficient

- Don't waste time on making your code efficient, until you know that that time will actually pay off.
- Knuth: 'premature optimization is the root of all evil'.
- ⇒ first make your code correct, then worry about efficiency

- Variables, functions, objects, form a new 'language':  
code in the language of the application.
- ⇒ your code should look like it talks about the application, not about memory.
- Levels of abstraction: implementation of a language should not be visible on the use level of that language.

#### 64.2 Programming: top-down versus bottom up

The exercises in chapter 45 were in order of increasing complexity. You can imagine writing a program that way, which is formally known as *bottom-up* programming.

However, to write a sophisticated program this way you really need to have an overall conception of the structure of the whole program.

Maybe it makes more sense to go about it the other way: start with the highest level description and gradually refine it to the lowest level building blocks. This is known as *top-down* programming.

<https://www.cs.fsu.edu/~myers/c++/notes/stepwise.html>

Example:

Run a simulation

becomes

Run a simulation:

    Set up data and parameters

    Until convergence:

        Do a time step

becomes

Run a simulation:

    Set up data and parameters:

        Allocate data structures

        Set all values

    Until convergence:

        Do a time step:

            Calculate Jacobian

            Compute time step

            Update

You could do these refinement steps on paper and wind up with the finished program, but every step that is refined could also be a subprogram.

We already did some top-down programming, when the prime number exercises asked you to write functions and classes to implement a given program structure; see for instance exercise 45.8.

A problem with top-down programming is that you can not start testing until you have made your way down to the basic bits of code. With bottom-up it's easier to start testing. Which brings us to...

#### 64.2.1 Worked out example

Take a look at exercise 6.14. We will solve this in steps.

1. State the problem:

```
// find the longest sequence
```

2. Refine by introducing a loop

```
// find the longest sequence:  
  
// Try all starting points  
// If it gives a longer sequence report
```

3. Introduce the actual loop:

```
// Try all starting points  
for (int starting=2; starting<1000; starting++) {  
// If it gives a longer sequence report  
}
```

4. Record the length:

```
// Try all starting points  
int maximum_length=-1;
```

```
for (int starting=2; starting<1000; starting++) {
    // If the sequence from 'start' gives a longer sequence report:
    int length=0;
    // compute the sequence from 'start'
    if (length>maximum_length) {
        // Report this sequence as the longest
    }
}
```

5. Refine computing the sequence:

```
// compute the sequence from 'start'
int current=starting;
while (current!=1) {
    // update current value
    length++;
}
```

6. Refine the update of the current value:

```
// update current value
if (current%2==0)
    current /= 2;
else
    current = 3*current+1;
```

## 64.3 Coding style

After you write your code there is the issue of *code maintainance*: you may in the future have to update your code or fix something. You may even have to fix someone else's code or someone will have to work on your code. So it's a good idea to code cleanly.

**Naming** Use meaningful variable names: `record_number` instead `rn` or `n`. This is sometimes called 'self-documenting code'.

**Comments** Insert comments to explain non-trivial parts of code.

**Reuse** Do not write the same bit of code twice: use macros, functions, classes.

## 64.4 Documentation

Take a look at Doxygen.

## 64.5 Best practices: C++ Core Guidelines

The C++ language is big, and some combinations of features are not advisable. Around 2015 a number of *Core Guidelines* were drawn up that will greatly increase code quality. Note that this is not about performance: the guidelines have basically no performance implications, but lead to better code.

## 64. Programming strategies

---

For instance, the guidelines recommend to use default values as much as possible when dealing with multiple constructors:

```
class Point { // not this way
private:
    double d;
public:
    Point( double x, double y, double fudge ) {
        auto d = ( x*x + y*y ) * (1+fudge); };
    Point( double x, double y ) {
        auto d = ( x*x + y*y ); };
};
```

This is bad because of code duplication. Slightly better:

```
class Point { // not this way
private:
    double d;
public:
    Point( double x, double y, double fudge ) {
        auto d = ( x*x + y*y ) * (1+fudge); };
    Point( double x, double y ) : Point(x,y,0.) {};
};
```

which wastes a couple of cycles if fudge is zero. Best:

```
class Point { // not this way
private:
    double d;
public:
    Point( double x, double y, double fudge=0. ) {
        auto d = ( x*x + y*y ) * (1+fudge); };
};
```

## Chapter 65

### Performance optimization

This section discusses performance and code optimization issues in the context of a random walk exercise.

#### 65.1 Problem statement

In 1904, Sir Ronald Ross, the biologist who discovered that malaria was carried by mosquitos, gave a lecture on ‘The Logical Basis of the Sanitary Policy of Mosquito Reduction’. In it, he considered the problem of how far a mosquito can fly, and therefore, how far away you need to drain all pools that harbor them.

We can model a mosquito as flying some unit distance in each time period, say, a day, of its life. However, since mosquitos fly in random directions, it will not cover a distance of  $N$  in the  $N$  days of its life. So how far does it get, statistically?

Ross was only able to compute this for a one-dimensional mosquito, that is, one that can only decide to go forward or backward along a line. In that case, the mosquito will on average get  $\sqrt{N}$  away from where it starts.

The more general problem was brought to mathematicians’ attention in 1905 by Karl Pearson, and turned out to have been solved in 1880 by Lord Rayleigh. This is now known as a *random walk* problem. Somewhat surprisingly, in 2D a mosquito also is expected to travel  $\sqrt{N}$ , where  $N$  is the number of time steps.

The general  $d$ -dimensional problem is a little harder, and the mosquito travels a little less than  $\sqrt{N}$ . Let’s code this in all generality.

#### 65.2 Coding

Main program setup:

**Code:**

```
// walk/walk_vec.cpp
float avg_dist{0.f};
for (int x=0; x<experiments; ++x) {
    Mosquito m(dim);
    for (int step=0; step<steps; ++step)
        m.step();
    avg_dist += m.distance();
}
avg_dist /= experiments;
```

**Output:**

[rand] vec:

```
D=3 after 10000 steps,
→distance= 83.7997
D=3 after 100000 steps,
→distance= 224.372
D=3 after 1000000 steps,
→distance= 922.599
product took: 2776
→milliseconds
```

where the `Mosquito` class stores its position:

```
// walk/walk_lib_vec.cpp
class Mosquito {
private:
    vector<float> pos;
public:
    Mosquito( int d )
        : pos( vector<float>(d,0.f) ) { };
```

and the `step` method updates this:

```
// walk/walk_lib_vec.cpp
void step() {
    int d = pos.size();
    auto incr = random_step(d);
    for (int id=0; id<d; ++id)
        pos.at(id) += incr.at(id);
}
```

The random step method produces a random coordinate, normalized to the unit circle. There is a slight problem here: if we generate a random coordinate in the unit cube, and normalize it, it will be biased towards the corners of the cube. Therefore, we iterate until we have a coordinate inside the unit circle, and use that to be normalized:

```
// rand/walk_lib_vec.cpp
vector<float> random_coordinate( int d ) {
    auto v = vector<float>(d);
    for ( auto& e : v )
        e = random_float();
    return v;
}

// walk/walk_lib_vec.cpp
vector<float> random_step(int d) {
    static std::random_device r;
    static std::default_random_engine static_engine( r() );
```

```

// Gaussian or normal distribution is spherically symmetric
static std::normal_distribution<float> gauss(2.f,1.f);
vector<float> step(d,gauss(static_engine));
auto norm = std::transform_reduce
    ( step.begin(),step.end(),step.begin(),
      0.f, std::plus<>(), std::multiplies<>() );
norm = std::sqrt(norm);
std::ranges::for_each( step, [norm] (auto c) { c/=norm; } );

return step;
} ;

```

**Exercise 65.1.** Take the basic code, and make a version based on

```

template<int d>
class Mosquito { /* ... */}

```

How much does this simplify your code? Do you get any performance improvement?

*You can base this off the file walk\_vec.cxx in the repository*

### 65.2.1 Optimization: save on allocation

Probably the main problem with this implementation is that each step creates multiple vectors. This sort of memory management is relatively costly, especially since very few operations are performed on each of these.

So we move the creation of the vectors outside of the computational routines. The random coordinates are now written into an array passed as parameter:

```

// walk/walk_lib_pass.cpp
void random_coordinate( vector<float>& v ) {
    for ( auto& e : v )
        e = random_float();
}

```

Likewise the random step:

```

// walk/walk_lib_pass.cpp
void random_step( vector<float>& step ) {
    for (;;) {
        random_coordinate(step);
    }
}

```

This process of passing the arrays in stops at the `step` method, which we want to keep parameterless. So we add an option `cache` to the constructor to store the step vector as well as the position:

**Code:**

```
// walk/walk_lib_pass.cpp
class Mosquito {
private:
    vector<float> pos;
    vector<float> inc;
    bool cache;
public:
    Mosquito( int d,bool cache=false )
        : pos( vector<float>(d,0.f) ),cache(
            cache) {
        if (cache) inc = vector<float>(d,0.f);
    };
}
```

**Output:**

```
[rand] pass:
D=3 after 10000 steps,
    ↪distance= 76.7711
D=3 after 100000 steps,
    ↪distance= 257.19
D=3 after 1000000 steps,
    ↪distance= 956.122
run took: 2852 milliseconds
D=3 after 10000 steps,
    ↪distance= 87.034
D=3 after 100000 steps,
    ↪distance= 256.655
D=3 after 1000000 steps,
    ↪distance= 912.033
run took: 1762 milliseconds
```

```
// walk/walk_lib_pass.cpp
void step() {
    int d = pos.size();
    if (cache) {
        random_step(inc);
        step( inc );
    } else {
        vector<float> incr(d);
        random_step(incr);
        step( incr );
    }
}
```

### 65.2.2 Caching in a static vector

There is still a problem with the `length` calculation. Since there is no reduction operator for ‘sum of squares’, we need to create a temporary vector for the squares, so that we can do a plus-reduction on it.

**Exercise 65.2.** Explore options for this temporary. Discuss what’s most elegant, and measure performance improvement.

- This temporary can be passed in as a parameter;
- it can be stored in a global variable;
- or we can declare it `static`.
- With the C++20 standard, you could also use the `ranges` header.

### 65.3 Vector vs array

In this simulation, we will mostly be working in 2D, so instead of using `vector`, we can use statically allocated `array` objects. This allows for the more elegantly functional interface:

```
// rand/walk_lib_arr.cpp
template<int d>
float length( const array<float,d>& step ) {
    array<float,d> square = step;
    for_each( square.begin(),square.end(),
        [] (float& x) { x *= x; } );
    auto l = sqrt
        ( std::accumulate( square.begin(),square.end(),0.f ) );
    return l;
};
```

While above we have removed all unnecessary allocation, we get an extra performance boost from optimizations from the compiler knowing the length of the array. Thus, instead of a loop of length two, the compiler will probably replace this by two explicit instructions.

**Code:**

```
// walk/walk_arr.cpp
float avg_dist{0.f};
for ( int x=0; x<experiments; ++x ) {
    Mosquito<dim> m;
    for ( int step=0; step<steps; ++step)
        m.step();
    avg_dist += m.distance();
}
avg_dist /= experiments;
```

**Output:**

```
[rand] arr:
D=3 after 10000 steps,
    ↪distance= 76.3221
D=3 after 100000 steps,
    ↪distance= 247.5
D=3 after 1000000 steps,
    ↪distance= 959.735
product took: 358
    ↪milliseconds
```



# Chapter 66

## Tiniest of introductions to algorithms and data structures

### 66.1 Data structures

The main data structure you have seen so far is the array. In this section we briefly sketch some more complicated data structures.

#### 66.1.1 Stack

A *stack* is a data structure that is a bit like an array, except that you can only see the last element:

- You can inspect the last element;
- You can remove the last element; and
- You can add a new element that then becomes the last element; the previous last element becomes invisible: it becomes visible again as the last element if the new last element is removed.

The actions of adding and removing the last element are known as *push* and *pop* respectively.

**Exercise 66.1.** Write a class that implements a stack of integers. It should have methods

```
void push(int value);
int pop();
```

#### 66.1.2 Linked lists

*Before doing this section, make sure you study section 16.*

Arrays are not flexible: you can not insert an element in the middle. Instead:

- Allocate a larger array,
- copy data over (with insertion),
- delete old array storage

This is expensive. (It's what happens in a C++ `vector`; section 10.3.2.)

If you need to do lots of insertions, make a *linked list*. The basic data structure is a *Node*, which contains

1. Information, which can be anything; and
2. A pointer (sometimes called ‘link’) to the next node. If there is no next node, the pointer will be *null*. Every language has its own way of denoting a *null pointer*; C++ has the `nullptr`, while C uses the `NULL` which is no more than a synonym for the value zero.

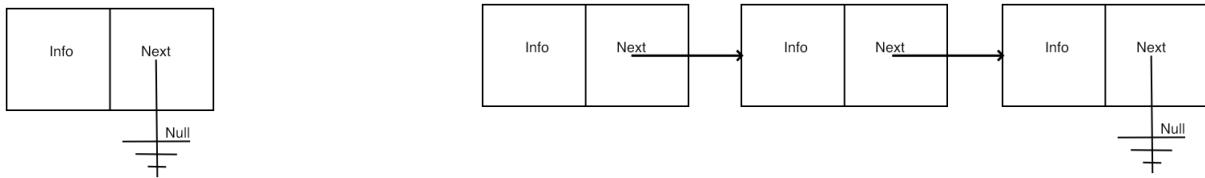


Figure 66.1: Node data structure and linked list of nodes

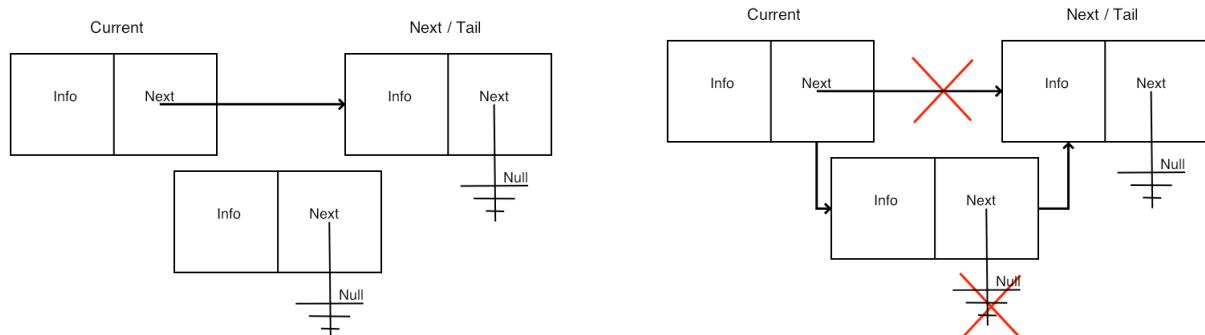


Figure 66.2: Insertion in a linked list

We illustrate this in figure 66.1.

Our main concern will be to implement operations that report some statistic of the list, such as its length, that test for the presence of information in the list, or that alter the list, for instance by inserting a new node. See figure 66.2.

#### 66.1.2.1 Data definitions

In C++ you have a choice of pointer types. For now, we will use `shared_ptr` throughout; later we will redo our code using `unique_ptr`.

We declare the basic classes. First we declare the `List` class, which has a pointer that is null for an empty list, and pointing to the first node otherwise.

A linked list has as its only member a pointer to a node:

```
// tree/linkshared.cpp
class List {
private:
    shared_ptr<Node> head{nullptr};
public:
    List() {};
```

Initially null for empty list.

Next, a `Node` has an information field, which we here choose to be an integer, and a counter to record how often a certain number has appeared. The `Node` also has a pointer to the next node. This pointer is again null for the last node in the list.

A node has information fields, and a link to another node:

```
// tree/linkshared.cpp
class Node {
private:
    int datavalue{0}, datacount{0};
    shared_ptr<Node> next{nullptr};
public:
    Node() {};
    Node(int value, shared_ptr<Node> next=nullptr)
        : datavalue(value), datacount(1), next(next) {};
```

A Null pointer indicates the tail of the list.

We are now going to develop methods for the *List* and *Node* classes that support the following code.

List testing and modification.

```
List mylist;
cout << "Empty list has length: "
    << mylist.length() << '\n';

mylist.insert(3);
cout << "After one insertion the length is: "
    << mylist.length() << '\n';
if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << '\n';
```

Let's start with some simple functions.

### 66.1.2.2 Simple functions

For many algorithms we have the choice between an iterative and a recursive version. The recursive version is easier to formulate, but the iterative solution is probably more efficient.

We start with a simple utility function for printing a linked list. (This function is somewhat crude; a better solution uses the strategy from section 12.6.) This implementation illustrates the recursive strategy.

Auxiliary function so that we can trace what we are doing.

Print the list head:

```
// tree/linkshared.cpp
void List::print() {
    cout << "List:" ;
    if (head!=nullptr)
        cout << " => "; head->print();
    cout << '\n';
};
```

Print a node and its tail:

```
// tree/linkshared.cpp
void Node::print() {
    cout << datavalue << ":" << datacount;
    if (has_next()) {
        cout << ", "; next->print();
    }
};
```

For recursively computing the length of a list, we adopt this same recursive scheme.

For the list:

```
// tree/linkshared.cpp
int List::length() {
    if (head==nullptr)
        return 0;
    else
        return head->length();
};
```

For a node:

```
// tree/linkshared.cpp
int Node::length() {
    if (!has_next())
        return 1;
    else
        return 1+next->length();
};
```

An iterative version uses a pointer that goes down the list, incrementing a counter at every step.

Use a shared pointer to go down the list:

```
// tree/linkshared.cpp
int List::length_iterative() {
    int count = 0;
    if (head!=nullptr) {
        auto current_node = head;
        while (current_node->has_next()) {
            current_node = current_node->nextnode(); count += 1;
        }
    }
    return count;
};
```

(Fun exercise: can do an iterative de-allocate of the list?)

**Exercise 66.2.** Write a function

```
bool List::contains_value(int v);
```

to test whether a value is present in the list.

This can be done recursive and iterative.

### 66.1.2.3 Modification functions

The interesting methods are of course those that alter the list. Inserting a new value in the list has basically two cases:

1. If the list is empty, create a new node, and set the head of the list to that node.
2. If the list is not empty, we have several more cases, depending on whether the value goes at the head of the list, the tail, somewhere in the middle. And we need to check whether the value is already in the list.

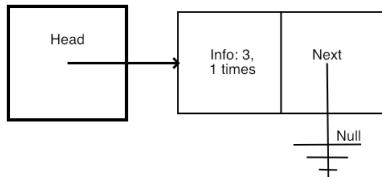
We will write functions

```
void List::insert(int value);
void Node::insert(int value);
```

that add the value to the list. The `List::insert` value can put a new node in front of the first one; the `Node::insert` assumes that the value is great equal that of the current node.

There are a lot of cases here. You can try this by an approach called Test-Driven Development (TDD): first you decide on a test, then you write the code that covers that case.

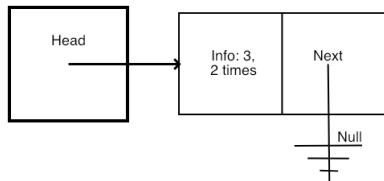
**Step 1: insert the first element** Adding a first element to an empty list is simple: we need the pointer of the head node to point to a `Node`.



**Exercise 66.3.** Next write the case of `Node::insert` that handles the empty list. You also need a method `List::contains` that tests if an item is in the list.

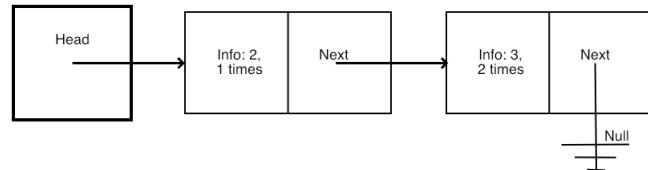
```
// tree/linkshared.cpp
mylist.insert(3);
cout << "After inserting 3 the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << '\n';
else
    cout << "Hm. Should contain 3" << '\n';
if (mylist.contains_value(4))
    cout << "Hm. Should not contain 4" << '\n';
else
    cout << "Indeed: does not contain 4" << '\n';
cout << '\n';
```

**Step 3: inserting an element that already exists** If we try to add a value to a list that is already there, inserting does not do anything; if needed we can increment a counter in the `Node` that contains that value.



**Exercise 66.4.** Inserting a value that is already in the list means that the `count` value of a node needs to be increased. Update your `insert` method to make this code work:

```
// tree/linkshared.cpp
mylist.insert(3);
cout << "Inserting the same item gives length: "
     << mylist.length() << '\n';
if (mylist.contains_value(3)) {
    cout << "Indeed: contains 3" << '\n';
    auto headnode = mylist.headnode();
    cout << "head node has value " << headnode->value()
        << " and count " << headnode->count() << '\n';
} else
    cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

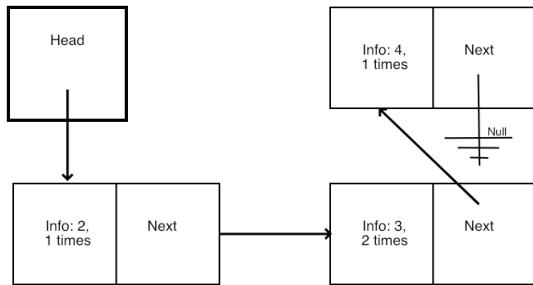


**Step 4: inserting an element at the head**

**Exercise 66.5.** One of the cases for inserting concerns an element that goes at the head. Update your `insert` method to get this to work:

```
// tree/linkshared.cpp
mylist.insert(2);
cout << "Inserting 2 goes at the head; now the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(2))
    cout << "Indeed: contains 2" << '\n';
else
    cout << "Hm. Should contain 2" << '\n';
if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << '\n';
else
    cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

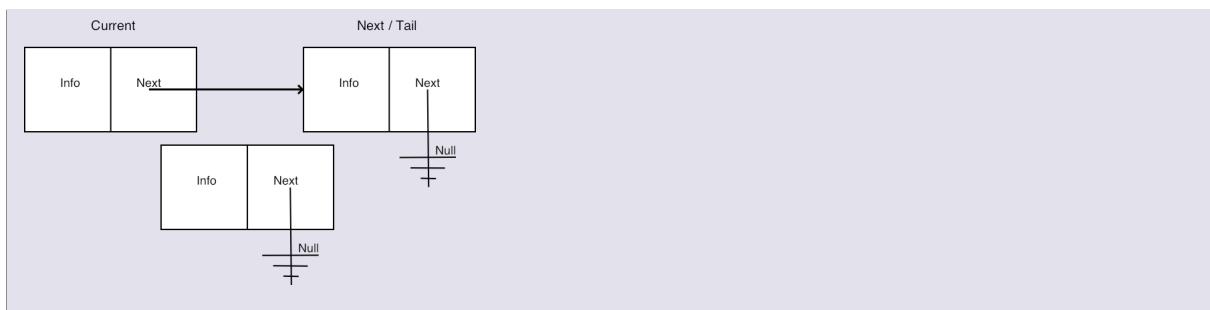
**Step 5: inserting an element at the end** Adding an element to the tail requires traversing the whole list.



**Exercise 66.6.** If an item goes at the end of the list:

```
// tree/linkshared.cpp
mylist.insert(6);
cout << "Inserting 6 goes at the tail; now the length is: "
     << mylist.length()
     << '\n';
if (mylist.contains_value(6))
    cout << "Indeed: contains 6" << '\n';
else
    cout << "Hm. Should contain 6" << '\n';
if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << '\n';
else
    cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

**Step 6: inserting an element in the middle** The trickiest case is inserting an element somewhere in the middle of the list. Now you need to compare the current and next element to decide whether to place the element or to move on to the tail.





**Exercise 66.7.** Update your insert routine to deal with elements that need to go somewhere in the middle.

```
// tree/linkshared.cpp
mylist.insert(4);
cout << "Inserting 4 goes in the middle; \nnow the length is: "
    << mylist.length()
    << '\n';
if (mylist.contains_value(4))
    cout << "Indeed: contains 4" << '\n';
else
    cout << "Hm. Should contain 4" << '\n';
if (mylist.contains_value(3))
    cout << "Indeed: contains 3" << '\n';
else
    cout << "Hm. Should contain 3" << '\n';
cout << '\n';
```

#### 66.1.2.4 Advanced: With unique pointers

Conceptually we can say that the list object owns the first node, and each node owns the next. Therefore, the most appropriate pointer type is the [unique\\_ptr](#).

We can also do this with unique pointers:

A linked list has as its only member a pointer to a node:

```
// tree/linkunique.cpp
class List {
private:
    unique_ptr<Node> head{nullptr};
public:
    List() {};
```

Initially null for empty list.

A node has information fields, and a link to another node:

```
// tree/linkunique.cpp
```

```

class Node {
    friend class List;
private:
    int datavalue{0}, datacount{0};
    unique_ptr<Node> next{nullptr};
public:
    friend class List;
    Node() {}
    Node(int value, unique_ptr<Node> tail=nullptr)
        : datavalue(value), datacount(1), next(move(tail)) {};
    ~Node() { cout << "deleting node " << datavalue << '\n'; };
}

```

A Null pointer indicates the tail of the list.

Above we formulated an iterative and a recursive way of computing the length of a list. The iterative code had a shared pointer pointing at successive list elements. We can not do this with unique pointers. Instead, this is a good place to use a *bare pointer*.

Use a *bare pointer*, which is appropriate here because it doesn't own the node.

```

// tree/linkunique.cpp
int listlength() {
    Node *walker = next.get();
    int len = 1;
    while ( walker!=nullptr ) {
        walker = walker->next.get();
        ++len;
    }
    return len;
}

// tree/linkshared.cpp
int List::length_iterative() {
    int count = 0;
    if (head!=nullptr) {
        auto current_node = head;
        while (current_node->has_next()) {
            current_node = current_node->next();
        }
    }
    return count;
}

```

(You will get a compiler error if you try to make `walker` a smart pointer: you can not copy a unique pointer.)

- Use smart pointers for ownership
- Use bare pointers for pointing but not owning.
- This is an efficiency argument.I'm not totally convinced.

### 66.1.3 Trees

*Before doing this section, make sure you study section 16.*

A tree can be defined recursively:

- A tree is empty, or

- a tree is a node with some number of children trees.

Let's design a tree that stores and counts integers: each node has a label, namely an integer, and a count value that records how often we have seen that integer.

Our basic data structure is the node, and we define it recursively to have up to two children. This is a problem: you can not write

```
class Node {  
private:  
    Node left,right;  
}
```

because that would recursively need infinite memory. So instead we use pointers.

```
// tree/binary.cpp  
class Node {  
private:  
    int key{0},count{0};  
    shared_ptr<Node> left,right;  
    bool hasleft{false},hasright{false};  
public:  
    Node() {}  
    Node(int i,int init=1) { key = i; count = 1; };  
    void addleft( int value) {  
        left = make_shared<Node>(value);  
        hasleft = true;  
    };  
    void addright( int value) {  
        right = make_shared<Node>(value);  
        hasright = true;  
    };  
    /* ... */  
};
```

and we record that we have seen the integer zero zero times.

Algorithms on a tree are typically recursive. For instance, the total number of nodes is computed from the root. At any given node, the number of nodes of that attached subtree is one plus the number of nodes of the left and right subtrees.

```
// tree/binary.cpp  
int number_of_nodes() {  
    int count = 1;  
    if (hasleft)  
        count += left->number_of_nodes();  
    if (hasright)  
        count += right->number_of_nodes();  
    return count;  
};
```

Likewise, the depth of a tree is computed as a recursive max over the left and right subtrees:

```
// tree/binary.cpp
int depth() {
    int d = 1, dl=0, dr=0;
    if (hasleft)
        dl = left->depth();
    if (hasright)
        dr = right->depth();
    d = max(d+dl,d+dr);
    return d;
};
```

Now we need to consider how actually to insert nodes. We write a function that inserts an item at a node. If the key of that node is the item, we increase the value of the counter. Otherwise we determine whether to add the item in the left or right subtree. If no such subtree exists, we create it; otherwise we descend in the appropriate subtree, and do a recursive insert call.

```
// tree/binary.cpp
void insert(int value) {
    if (key==value)
        ++count;
    else if (value<key) {
        if (hasleft)
            left->insert(value);
        else
            addleft(value);
    } else if (value>key) {
        if (hasright)
            right->insert(value);
        else
            addright(value);
    } else throw(1); // should not happen
};
```

#### 66.1.4 Other graphs

The nodes in a tree have a relation from parent-to-child, so they are a special case of a *directed graph*.

**Exercise 66.8.** If you know about the relationship between graphs and their *adjacency matrix*, can you express directedness in terms of matrix properties?

An important subset of directed graphs, that of DAGs, has a property that trees do not have: for some pairs of nodes they may have more than one path between them.

For more details, see HPC book [13], chapter 25.

## 66.2 Algorithms

This **really really** goes beyond this book.

- Simple ones: numerical
- Connected to a data structure: search

### 66.2.1 Sorting

Unlike the tree algorithms above, which used a non-obvious data structure, sorting algorithms are a good example of the combination of very simple data structures (mostly just an array), and sophisticated analysis of the algorithm behavior. We very briefly discuss two algorithms.

The standard library has a sorting routine built-in; see section [14.3.6](#).

#### 66.2.1.1 Bubble sort

An array  $a$  of length  $n$  is sorted if

$$\forall_{i < n-1} : a_i \leq a_{i+1}.$$

A simple sorting algorithm suggests itself immediately: if  $i$  is such that  $a_i > a_{i+1}$ , then reverse the  $i$  and  $i + 1$  locations in the array.

```
// sort/bubble.cpp
void swapij( vector<int> &array, int i ) {
    int t = array[i];
    array[i] = array[i+1];
    array[i+1] = t;
}
```

(Why is the array argument passed by reference?)

If you go through the array once, swapping elements, the result is not sorted, but at least the largest element is at the end. You can now do another pass, putting the next-largest element in place, and so on.

This algorithm is known as *bubble sort*. It is generally not considered a good algorithm, because it has a time complexity (section [70.2](#)) of  $n^2/2$  swap operations. Sorting can be shown to need  $O(n \log n)$  operations, and bubble sort is far above this limit.

#### 66.2.1.2 Quicksort

A popular algorithm that can attain the optimal complexity (but need not; see below) is *quicksort*:

- Find an element, called the pivot, that is approximately equal to the median value.
- Rearrange the array elements to give three sets, consecutively stored: all elements less than, equal, and greater than the pivot respectively.
- Apply the quicksort algorithm to the first and third subarrays.

This algorithm is best programmed recursively, and you can even make a case for its parallel execution: every time you find a pivot you can double the number of active processors.

**Exercise 66.9.** Suppose that, by bad luck, your pivot turns out to be the smallest array element every time. What is the time complexity of the resulting algorithm?

### 66.2.2 Graph algorithms

We briefly discuss some graph algorithms. For further discussion see HPC book [13], chapter 10.

First consider the SSSP problem: given a graph and a starting node, what is the shortest path to every other node. Initially, we consider an *unweighted graph*, or equivalently, set the distance between any pair of connected nodes to 1.

The algorithm proceeds by levels: each next level consists of the neighbors of already explored nodes.

- Set the distance to the starting node to zero.
- Until done,
- Loop over all nodes with known distance, and
- for all of its neighbors,
  - unless the neighbor already has a known distance
  - set the distances to  $d + 1$ ,

We use a simple data structure for the distances:

```
map<int, int> distances;
int starting_node = 0;
distances[starting_node] = 0;
```

Until all nodes are mapped, we iterate over nodes for which the distances are known, and set the distance for their neighbors:

```
// while not done
for (;;) {
    int updates{0};
    // iterate over all mapped nodes
    for ( auto [node,dist] : distances ) {
        // for each, iterate over all of their neighbors
    }
}
```

Assume that the graph data structure supports getting the list of neighbors of a node:

```
for ( const auto& neighbor : graph.neighbors(node) ) {
    if ( auto find_neighbor = distances.find(neighbor) ; find_neighbor==distances.end()
        () ) {
        distances[neighbor] = dist+1; updates++;
    }
}
```

Since we use a until-done loop, we need to break out of explicitly. A simple test would be ‘if the number of mapped nodes equals the number of nodes in the graph’.

**Exercise 66.10.** Can you see a problem with this, and a way to fix it?

The above algorithm is a little wasteful: in each pass it traverses all mapped nodes, but we only need the newly added ones. Therefore we add:

```
set<int> current_front;
current_front.insert(starting_node);
for (;;) {
    set<int> next_front;
    // iterate over current_front
    for ( auto node : current_front ) {
        // iterate over neighbors
        for ( const auto& neighbor : graph.neighbors(node) ) {
            // if neighbor not yet mapped
            if ( /* ... */ ) {
                distances[neighbor] = dist_to_this_node+1;
                next_front.insert(neighbor);
            }
        }
    }
    current_front = next_front;
}
```

## 66.3 Programming techniques

### 66.3.1 Memoization

In section 7.6 you saw some examples of recursion. The factorial example could be written in a loop, and there are both arguments for and against doing so.

The Fibonacci example is more subtle: it can not immediately be converted to an iterative formulation, but there is a clear need for eliminating some waste that comes with the simple recursive formulation. The technique we can use for this is known as *memoization*: store intermediate results to prevent them from being recomputed.

Here is an outline.

```
// recur/fibomemo.cpp
int fibonacci(int n) {
    vector<int> fibo_values(n);
    for (int i=0; i<n; ++i)
        fibo_values[i] = 0;
    fibonacci_memoized(fibo_values, n-1);
    return fibo_values[n-1];
}
int fibonacci_memoized( vector<int> &values, int top ) {
    int minus1 = top-1, minus2 = top-2;
    if (top<2)
        return 1;
    if (values[minus1]==0)
        values[minus1] = fibonacci_memoized(values,minus1);
    if (values[minus2]==0)
        values[minus2] = fibonacci_memoized(values,minus2);
    values[top] = values[minus1]+values[minus2];
    //cout << "set f(" << top << ")" to " << values[top] << '\n';
```

```
    return values[top];  
}
```



## Chapter 67

### Provably correct programs

Programming often seems more an art, even a black one, than a science. Still, people have tried systematic approaches to program correctness. One can distinguish between

- proving that a program is correct, or
- writing a program so that it is guaranteed to be correct.

This distinction is only imaginary. A more fruitful approach is to let the proof drive the coding. As E.W. Dijkstra pointed out

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.

We will see a couple of examples of this.

#### 67.1 Loops as quantors

Quite often, algorithms can be expressed mathematically. In that case you should make your program look like the mathematics. Here we consider one example, pointing out the connection between for-loops and mathematical quantors.

##### 67.1.1 Forall-quantor

Consider a simple example: testing if a number is prime. The predicate ‘*isprime*’ can be expressed as:

$$\text{isprime}(n) \equiv \forall_{2 \leq f < n} : \neg \text{divides}(f, n)$$

You see that proving the original *isprime* predicate for some value  $n$  now involves

1. a quantor over a new variable  $f$  – we call this a ‘bound variable’;
2. and a new predicate `divides` that involves the original variable  $n$  and the variable  $f$  that is bound by the quantor.

We now spell out the ‘for all’ quantor iteratively as a loop where each iteration needs to be true. That is, we do an ‘and’ reduction on some iteration-dependent result.

$$\neg \text{divides}(2, n) \cap \dots \cap \neg \text{divides}(n - 1, n)$$

And this sequence of ‘and’ conjunctions can be programmed:

```
for (int f=2; f<n; f++)
    isprime = isprime && not divides(f, p)
```

You notice that the loop variable is the variable  $f$  that was introduced by the quantor.

Now our only worry is how to initialize `isprime`. The initial value corresponds to an ‘and’ conjunction over an empty set, which is true, so:

```
bool isprime{true};
for (int f=2; f<n; f++)
    isprime = isprime && not divides(f, p)
```

**Exercise 67.1.** Now that you have a loop that computes the right thing you can start worrying about performance. Can the loop be terminated prematurely in some cases? How would you program that?

### 67.1.2 Thereis-quantor

What if we had expressed primeness as:

$$\text{isprime}(n) \equiv \neg \exists_{2 \leq f < n} : \text{divides}(f, n)$$

To get a pure quantor, and not a negated one, we write:

$$\text{isnotprime}(n) \equiv \exists_{2 \leq f < n} : \text{divides}(f, n)$$

Spelling out the exists-quantor as

$$\text{isnotprime}(n) \equiv \text{divides}(2, n) \cup \dots \cup \text{divides}(n - 1, n)$$

we see that we need a loop where we test if any iteration satisfies a predicate. That is, we do an ‘or’-reduction on the results of each iteration. As before, the loop variable is the variable introduced by the quantor.

```
for (int f=2; f<n; f++)
    isnotprime = isnotprime or divides(f, p)
bool isprime = not isnotprime;
```

Also as before, we take care to initialize the reduction variable correctly: applying  $\exists_{s \in S} P(s)$  over an empty set  $S$  is `false`:

```
bool isnotprime{false};
for (int f=2; f<n; f++)
    isnotprime = isnotprime or divides(f, p)
bool isprime = not isnotprime;
```

**Exercise 67.2.** Same question as for the ‘forall’ quantor: can this loop be terminated prematurely? How would you code that?

### 67.1.3 Quantors through ranges

We have the following correspondences:

```
∀ all_of
∃ any_of
```

Let  $S$  be a range and  $P$  a predicate, then

$$\forall_{n \in S} : P(n)$$

can be implemented as

```
all_of( S, [] { auto n } -> bool { return P(n); } );
```

Likewise,

$$\exists_{n \in S} : P(n)$$

can be implemented as

```
any_of( S, [] { auto n } -> bool { return P(n); } );
```

In the following example we ‘prove’ a simple statement over integers. Using `iota(1)` to generate all integers would lead to infinite runtime, so we truncate with a second parameter.

Example:

```
∀_n: ∃_m: m > n

// primes/rangepred.cpp
all_of( rng::views::iota(1, 20),
        [] ( auto n ) -> bool {
            return any_of( rng::views::iota(1),
                           [n] ( auto m ) -> bool {
                               if (m>n) {
```

Ranging over more complicated sets:

$$\begin{aligned} \forall & \ n \in S : P(n) \\ & Q(n) \end{aligned}$$

can be implemented as

```
all_of( S | filter( [] (auto n) -> bool { return Q(n); } ),
        [] { auto n } -> bool { return P(n); } );
```

**Exercise 67.3.** Write range-based code that proves

$$\forall_n : \exists_{m>n} : \text{even}(m)$$

If you're doing the prime numbers project, you can now do exercise 45.16.

## 67.2 Predicate proving

For programs that have a clear loop structure you can take an approach that is similar to doing a ‘proof by induction’.

Let us consider the Collatz conjecture again, where for brevity we define

$$c(\ell) = \text{the length of the Collatz sequences, starting on } \ell.$$

Now we consider the Collatz conjecture as proving a predicate

$$P(\ell_k, m_k, k) = \begin{cases} \ell_k < k \text{ is the location of the longest sequence:} \\ c(\ell_k) = m_k: \text{the length of sequence } \ell_k \text{ is } m_k \\ \text{all other sequences } \ell < k \text{ are shorter} \end{cases}$$

Formally:

$$P(\ell_k, m_k, k) = \left[ \begin{array}{l} \ell_k < k \\ \wedge \quad c(\ell_k) = m_k (\text{only if } k > 0) \\ \wedge \quad \forall_{\ell < k} : c(\ell) \leq m_k \end{array} \right]$$

for  $k = N$ .

We develop the code that makes this predicate inductively true. We start out with

$$\ell_0 = -1, \quad m_0 = 0 \Rightarrow P(\ell_0, m_0, 0).$$

The inductive proof corresponds to a loop:

- we assume that at the start of the  $k$ -th iteration  $P(\ell_k, m_k, k)$  is true;
- the iteration body is such that at the end of the  $k$ -th iteration  $P(\ell_{k+1}, m_{k+1}, k + 1)$  is true;
- this of course sets up the predicate at the start of the next iteration.

The loop structure is then:

```
k=0;
{P(lk, mk, k)}
while ( k < N ) {
    {P(lk, mk, k)}
    update;
    {P(lk+1, mk+1, k + 1)}
    k = k+1;
}
```

The update has to extend the predicate from  $k$  to  $k + 1$ . Let us consider the parts of it.

We need to establish

$$\forall_{\ell < k+1} : c(\ell) \leq m_{k+1}$$

We split the range  $\ell < k + 1$  into  $\ell < k$  and  $\ell = k$ :

- the first part

$$\forall_{\ell < k} : c(\ell) \leq m_{k+1}$$

is true if  $m_{k+1} \geq m_k$ ;

- the part

$$\ell = k : c(\ell) \leq m_{k+1}$$

states that  $m_{k+1} \geq c(k)$ .

Together we get that

$$m_{k+1} \geq \max(m_k, c(k))$$

Finally, the clause

$$c(\ell_{k+1}) = m_{k+1}$$

can be satisfied:

- If  $c(k) > m_k$ , we need to set  $m_{k+1} = c(k)$  and  $\ell_{k+1} = k$ .
- (Strictly speaking, there is a possibility  $m_{k+1} > c(k)$ . This is not possible, because we can not satisfy  $m_{k+1} = c(\ell_k)$  for any  $k$ .)
- If  $c(k) \leq m_k$ , we need to set  $m_{k+1} \geq m_k$ . Again,  $m_{k+1} > m_k$  can not be satisfied by any  $\ell_{k+1}$ , so we conclude  $m_{k+1} = m_k$ .

## 67.3 Flame

Above, you saw a Dijkstra quote where he argues that testing is insufficient to show correctness of a program. So how does Dijkstra envision that correctness can be ensured? That can be found in the second part of his quote:

The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.

Let us develop this though, using matrix-vector multiplication as a simple example: we will derive the algorithm hand-in-hand with its correctness proof.

Many linear algebra algorithms are loop-based, and the foundation to a correctness proof is the derivation of a *loop invariant*: a predicate that is inductively shown to be true in each loop iteration, thereby guaranteeing the correctness of the whole algorithm.

### 67.3.1 Derivation of the common algorithm

As a preliminary to deriving a loop invariant, we first consider the result of the computation in a partitioned form.

$$y = Ax$$

Partitioned:

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Two equations:

$$\begin{cases} y_T = A_T x \\ y_B = A_B x \end{cases}$$

The key to the inductive proof is to take this partitioned form, and assume that it is only partly satisfied.

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Assume only equation

$$y_T = A_T x$$

is satisfied.

Now we are getting close to an inductive proof: we consider the algorithm as aimed at increasing the size of the block for which the predicate is true. If this block equals the size of the problem, we have correctness of the full result.

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

While  $T$  is not the whole system

Predicate:  $y_T = A_T x$  true

Update: grow  $T$  block by one

Predicate:  $y_T = A_T x$  true for new/bigger  $T$  block

Note initial and final condition.

Now we compare the true statement in one iteration, and that in the next, and compare the two blocks for which the predicate holds.

Here is the big trick

Before

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

split:

$$\begin{pmatrix} y_1 \\ \dots \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \dots \\ A_2 \\ A_3 \end{pmatrix} (x)$$

Then the update step, and

After

$$\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \\ \dots \\ A_3 \end{pmatrix} (x)$$

and unsplit

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Comparing these two blocks gives us the extra predicate that was made to be satisfied by the iteration we consider:

Before the update:

$$\begin{pmatrix} y_1 \\ \dots \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \dots \\ A_2 \\ A_3 \end{pmatrix} (x)$$

so

$$y_1 = A_1 x$$

is true

Then the update step, and

After

$$\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \\ \dots \\ A_3 \end{pmatrix} (x)$$

so

$$\begin{cases} y_1 = A_1x & \text{we had this} \\ y_2 = A_2x & \text{we need this} \end{cases}$$

This extra predicate can trivially be converted to an elementary computation, and so we find the full algorithm:

While  $T$  is not the whole system  
 Predicate:  $y_T = A_Tx$  true  
 Update:  $y_2 = A_2x$   
 Predicate:  $y_T = A_Tx$  true for new/bigger  $T$  block

#### 67.3.1.1 Derivation of the algorithm by columns

In the previous section we derived the matrix-vector product, expressed the usual way: each element of the output vector is the inner product of a matrix row and the input vector. You may think that this is a lot of to-do for not much result.

Consider then that we can derive other algorithms for the matrix-vector product, using the same general principle. Our basic assumption was that we split the matrix horizontally in two block rows. What would happen if we split the matrix vertically in two block columns?

We divide the matrix into two block columns, and express the result of the matrix-vector product on this form.

$$y = Ax$$

Partitioned:

$$(y) = (A_L \quad A_R) \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

Equation:

$$\begin{cases} y = A_Lx_T + A_Rx_B \end{cases}$$

Again we make a statement about a partially completed computation.

$$(y) = (A_L \quad A_R) \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

Assume

$$y = A_Lx_T$$

is constructed, and grow the  $T$  block.

Again we compare splitting the matrix in one iteration and the next, and comparing the partial predicates:

Before

$$(y) = (A_L \quad A_R) \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

split:

$$(y) = \begin{pmatrix} A_1 & \vdots & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ \cdots \\ x_2 \\ x_3 \end{pmatrix}$$

Then the update step, and

After

$$(y) = \begin{pmatrix} A_1 & A_2 & \vdots & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_3 \end{pmatrix}$$

and unsplit

$$(y) = (A_L \quad A_R) \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

This gives us by ‘subtracting’ one predicate from the other the extra result that was derived by the single iteration, and therefore the computation that was done in that iteration.

Before the update:

$$(y) = \begin{pmatrix} A_1 & \vdots & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ \cdots \\ x_2 \\ x_3 \end{pmatrix}$$

so

$$y = A_1 x_1$$

is true

Then the update step, and

After

$$(y) = \begin{pmatrix} A_1 & A_2 & \vdots & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_3 \end{pmatrix}$$

so

$$y = A_1 x_1 + A_2 x_2$$

in other words, we need

$$y \leftarrow y + A_2 x_2$$

As a result we obtain a second form of the matrix-vector product.

While  $T$  is not the whole system

Predicate:  $y = A_L x_T$  true

Update:  $y \leftarrow y + A_2 x_2$

Predicate:  $y = A_L x_T$  true for new/bigger  $T$  block

In quasi-Matlab notation we express both algorithms:

for  $r = 1, m$

$$y_r = A_{r,*} x_*$$

$$y \leftarrow 0$$

for  $c = 1, n$

$$y \leftarrow y + A_{*,c} x_c$$

## Chapter 68

### Unit testing and Test-Driven Development

In an ideal world, you would prove your program correct, but in practice that is not always feasible, or at least: not done. Most of the time programmers establish the correctness of their code by testing it.

Yes, there is a quote by *Edsger Dijkstra* that goes:

Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. (cue laughter)

but that doesn't mean that you can't at least gain some confidence in your code by testing it.

#### 68.1 Types of tests

Testing code is an art, even more than writing the code to begin with. That doesn't mean you can't be systematic about it. First of all, we distinguish between some basic types of test:

- *Unit tests* that test a small part of a program by itself;
- *System tests* test the correct behavior of the whole software system; and
- *Regression tests* establish that the behavior of a program has not changed by adding or changing aspects of it.

In this section we will talk about unit testing.

A program that is written in a sufficiently modular way allows for its components to be tested without having to wait for an all-or-nothing test of the whole program. Thus, testing and program design are aligned in their interests. In fact, writing a program with the thought in mind that it needs to be testable can lead to cleaner, more modular code.

In an extreme form of this you would write your code by Test-Driven Development (TDD), where code development and testing go hand-in-hand. The basic principles can be stated as follows:

- Both the whole code and its parts should always be testable.
- When extending the code, make only the smallest change that allows for testing.
- With every change, test before and after.
- Assure correctness before adding new features.

In a strict interpretation, you would even for each part of the program first write the test that it would satisfy, and then the actual code.

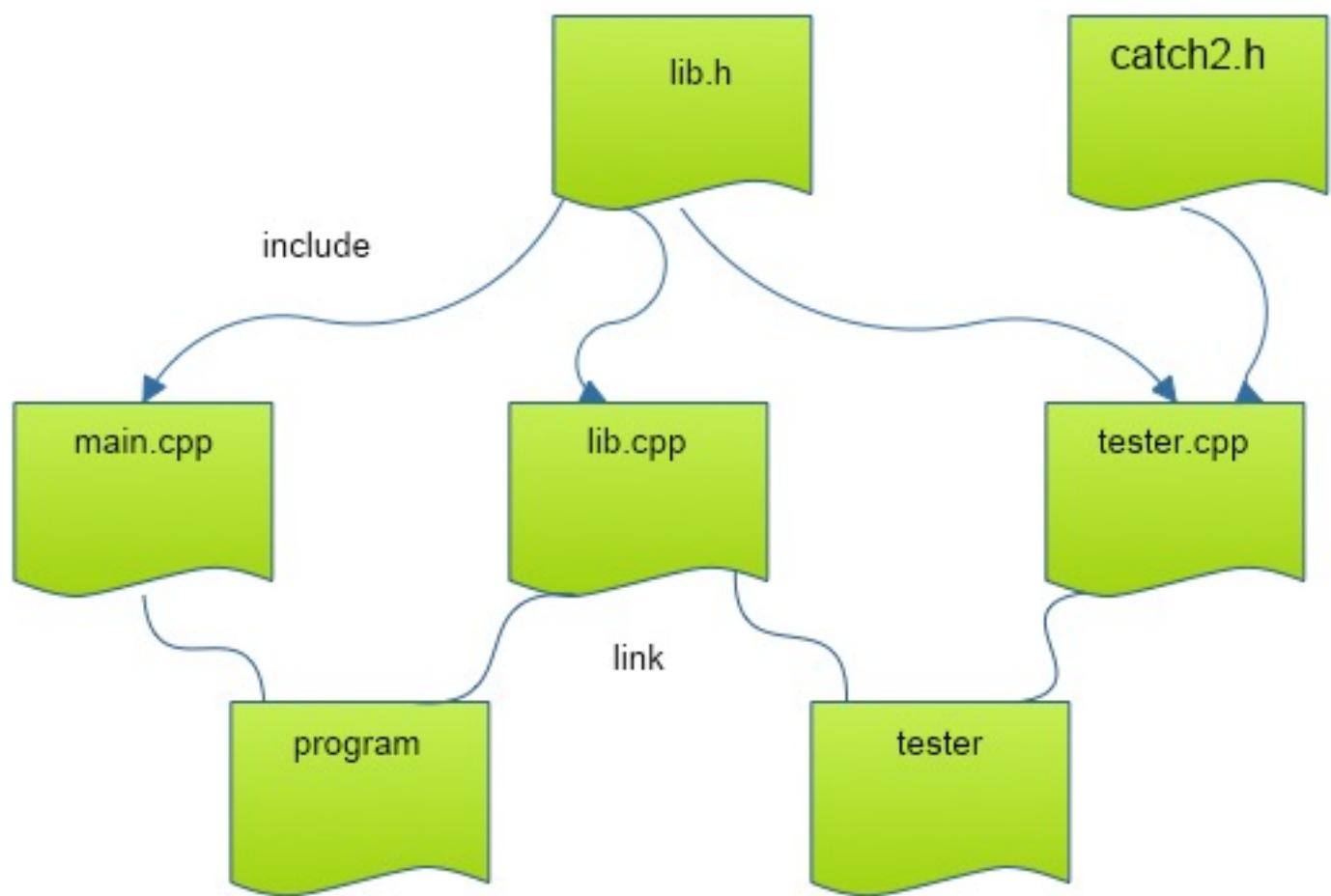


Figure 68.1: File structure for unit tests

## 68.2 Unit testing frameworks

There are several ‘frameworks’ that help you with unit testing. In the remainder of this chapter we will use *Catch2*, which is one of the most used ones in C++.

See section 68.6 for the practical matters of how to obtain, install, and compile with Catch2.

### 68.2.1 Test cases

A test case is a short program that is run as an independent main. In the setup suggested above, you put all your unit tests in the tester main program, that is, the file that has the

```
#define CATCH_CONFIG_MAIN  
#include "catch2/catch_all.hpp"
```

magic lines.

Each test case needs to have a unique name, which is printed when a test fails. You can optionally add keys to the test case that allow you to select tests from the commandline.

```
TEST_CASE( "name of this test" ) {
    // stuf
}
TEST_CASE( "name of this test", "[key1][key2]" ) {
    // stuf
}
```

The body of the test case is essentially a main program, where some statements are encapsulated in test macros. The most common macro is `REQUIRE`, which is used to demand correctness of some condition.

Tests go in `tester.cpp`:

```
TEST_CASE( "test that f always returns positive" ) {
    for (int n=0; n<1000; n++)
        REQUIRE( f(n)>0 );
}
```

- `TEST_CASE` acts like independent main program.  
can have multiple cases in a tester file
- `REQUIRE` is like `assert` but more sophisticated

### Exercise 68.1.

1. Write a function

```
double f(int n) { /* .... */ }
```

that takes on positive values only.

2. Write a unit test that tests the function for a number of values.

You can base this off the file `tdd.cxx` in the repository

Boolean:

```
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
```

Integer:

```
REQUIRE( integer_function(1)==3 );
REQUIRE( integer_function(1)!=0 );
```

Boolean expressions need to be parenthesized:

```
REQUIRE( ( x>0 and x<1 ) );
```

For failing tests, the framework will give the name of the test, the line number, and the values that were tested.

Run the tester:

```
-----
test the increment function
-----
test.cpp:25
```

```
.....
test.cpp:29: FAILED:
    REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
    1 == 2

=====
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

In the above case, the error message printed out the offending value of  $f(n)$ , not the value of  $n$  for which it occurs. To determine this, insert `INFO` specifications, which only get print out if a test fails.

```
INFO: print out information at a failing test
TEST_CASE( "test that f always returns positive" ) {
    for (int n=0; n<1000; n++)
        INFO( "iteration: " << n );
    REQUIRE( f(n)>0 );
}
```

If your code throws exceptions (section 23.2) you can test for these.

Suppose function  $g(n)$

- succeeds for input  $n > 0$
- fails for input  $n \leq 0$ :  
throws exception

```
TEST_CASE( "test that g only works for positive" ) {
    for (int n=-100; n<+100; n++)
        if (n<=0)
            REQUIRE_THROWS( g(n) );
        else
            REQUIRE_NO_THROW( g(n) );
}
```

### 68.2.1.1 Floating point tests

As a general guideline, one should never test equality of floating point numbers. In the case of unit testing that means that we need a test with some amount of leeway.

```
REQUIRE( real_function(1.5)==Catch::Approx(3.0) );
REQUIRE( real_function(1)!=Catch::Approx(1.0) );
REQUIRE( zero_find()==Catch::Approx(0.).margin(1.e-8) );
```

### 68.2.1.2 Commonalities

A common occurrence in unit testing is to have multiple tests with a common setup or tear down, to use terms that you sometimes come across in unit testing. Catch2 supports this: you can make ‘sections’ for part in between setup and tear down.

Use SECTION if tests have intro/outtro in common:

```
TEST_CASE( "commonalities" ) {
    // common setup:
    double x, y, z;
    REQUIRE_NOTHROW( y = f(x) );
    // two independent tests:
    SECTION( "g function" ) {
        REQUIRE_NOTHROW( z = g(y) );
    }
    SECTION( "h function" ) {
        REQUIRE_NOTHROW( z = h(y) );
    }
    // common followup
    REQUIRE( z > x );
}
```

(sometimes called setup/teardown)

### 68.3 Example: zero-finding by bisection

Development of the zero-finding algorithm by bisection can be found in section [47.1](#).

### 68.4 An example: quadratic equation roots

We revisit exercise [24.9](#), which used `std::variant` to return 0,1,2 roots of a quadratic equation. Here we use TDD to arrive at the code.

Throughout, we represent the polynomial

$$ax^2 + bx + c$$

as

```
using quadratic = tuple<double, double, double>;
```

When needed, you can unpack this again with

```
auto [a, b, c] = coefficients;
```

(Can you think of an `assert` statement here that might be useful?)

**Exercise 68.2.** Write a function

```
double discriminant( quadratic coefficients );
```

that computes  $b^2 - 4ac$ , and test:

```
// union/quadtest.cpp
TEST_CASE( "discriminant" ) {
    quadratic one{0., 2.5, 0.};
    REQUIRE( discriminant( one ) == Catch::Approx(6.25) );
    quadratic two{1., 0., 1.5};
    REQUIRE( discriminant( two ) == Catch::Approx(-6.) );
```

```
quadratic three{.1, .1, .1*.5};  
REQUIRE( discriminant( three ) ==Catch::Approx(-.01) );  
}
```

It may be illustrative to see what happens if you leave out the approximate equality test:

```
REQUIRE( discriminant( make_tuple(.1, .1, .1*.5) ) == -.01 );
```

With this function it becomes easy to detect the case of no roots: the discriminant  $D < 0$ . Next we need to have the criterium for single or double roots: we have a single root if  $D = 0$ .

**Exercise 68.3.** Write a function

```
bool discriminant_zero( quadratic coefficients );
```

that passes the test

```
// union/quadtest.cpp  
quadratic coefficients{a,b,c};  
d = discriminant( coefficients );  
z = discriminant_zero( coefficients );  
INFO( a << "," << b << "," << c << " d=" << d );  
REQUIRE( z );
```

Using for instance the values:

```
a = 2; b = 4; c = 2;  
a = 2; b = sqrt(40); c = 5; //!!!  
a = 3; b = 0; c = 0.;
```

This exercise is the first one where we run into numerical subtleties. The second set of test values has the discriminant zero in exact arithmetic, but nonzero in computer arithmetic. Therefore, we need to test whether it is small enough, compared to  $b$ .

**Exercise 68.4.** Be sure to also test the case where `discriminant_zero` returns false.

Now that we've detected a single root, we need the function that computes it. There are no subtleties in this one.

**Exercise 68.5.** Write the function `simple_root` that returns the single root. For confirmation, test

```
// union/quadtest.cpp  
auto r = simple_root(coefficients);  
REQUIRE( evaluate(coefficients,r)==Catch::Approx(0.).margin(1.e-14) );
```

The remaining case of two distinct roots is arrived at by elimination, and the only thing to do is write the function that returns them.

**Exercise 68.6.** Write a function that returns the two roots as a `indexcstdpair`:

```
pair<double,double> double_root( quadratic coefficients );
```

Test:

```
// union/quadtest.cpp
quadratic coefficients{a,b,c};
auto [r1,r2] = double_root(coefficients);
auto
e1 = evaluate(coefficients,r1),
e2 = evaluate(coefficients,r2);
REQUIRE( evaluate(coefficients,r1)==Catch::Approx(0.).margin(1.e-14) );
REQUIRE( evaluate(coefficients,r2)==Catch::Approx(0.).margin(1.e-14) );
```

The final bit of code is the function that tests for how many roots there are, and returns them as a `std::variant`.

### Exercise 68.7. Write a function

```
variant< bool,double, pair<double,double> >
compute_roots( quadratic coefficients );
```

Test:

```
// union/quadtest.cpp
TEST_CASE( "full test" ) {
    double a,b,c; int index;
    SECTION( "no root" ) {
        a=2.0; b=1.5; c=2.5;
        index = 0;
    }
    SECTION( "single root" ) {
        a=1.0; b=4.0; c=4.0;
        index = 1;
    }
}

SECTION( "double root" ) {
    a=2.2; b=5.1; c=2.5;
    index = 2;
}
quadratic coefficients{.a=a,.b=b,.c=c
};
auto result = compute_roots(
coefficients);
REQUIRE( result.index()==index );
```

## 68.5 Eight queens example

See 48.3.

## 68.6 Practical aspects of using Catch2

### 68.6.1 Installing Catch2

You can find the code for Catch2 at <https://github.com/catchorg>. You can either downloading a release version, or clone the repository. Here I am assuming version 3.1.1.

```
// download release version
wget https://github.com/catchorg/Catch2/archive/refs/tags/v3.1.1.tar.gz
// unpack
tar fxz v3.1.1.tar.gz
// make build and install directories
mkdir -p build-catch2
mkdir -p installation-catch2
// go to the build directory
cd build
```

```
// build the installation
cmake -D CMAKE_INSTALL_PREFIX=../installation-catch2 \
      -D BUILD_SHARED_LIBS=TRUE \
      ../Catch2-3.1.1
make
make install
```

### 68.6.2 Two usage modes

Let's assume you have a file structure with

- a very short main program, and
- a library file that has all functions used by the main.

In order to test the functions, you supply another main, which contains only unit tests; This is illustrated in figure 68.1.

In fact, with Catch2 your main file doesn't actually have a `main` program: that is supplied by the framework. In the tester main file you only put the test cases.

The framework supplies its own main:

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

#include "library_functions.h"
/*
 here follow the unit tests
*/
```

One important question is what header file to include. You can do

```
#include "catch.hpp"
```

which is the ‘header only’ mode, but that makes compilation very slow. Therefore, we will assume you have installed Catch through *Cmake*, and you include

```
#include "catch2/catch_all.hpp"
```

Note: as of September 2021 this requires the development version of the repository, not any 2 . x release.

### 68.6.3 Compilation

The setup suggested above requires you to add compile and link flags to your setup. This is system-dependent.

One-line solution:

```
icpc -o tester test_main.cpp \
      -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
      -lCatch2Main -lCatch2
```

**Variables for a Makefile:**

```
INCLUDES = -I${TACC_CATCH2_INC}
EXTRALIBS = -L${TACC_CATCH2_LIB} -lCatch2Main -lCatch2
```

In CMake, Catch2 can be discovered through *pkgconfig*:

**Lines for a CMake configuration:**

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( CATCH2 REQUIRED catch2-with-main )
target_include_directories(
    ${PROGRAM_NAME} PUBLIC
    ${CATCH2_INCLUDE_DIRS}
)
target_link_directories(
    ${PROGRAM_NAME} PUBLIC
    ${CATCH2_LIBRARY_DIRS}
)
```



# Chapter 69

## Debugging with gdb

### 69.1 A simple example

The following program does not have any bugs; we use it to show some of the basics of gdb.

```
// gdb/hello.cpp
void say(int n) {
    cout << "hello world " << n << '\n';
}

int main() {
    for (int i=0; i<10; ++i) {
        int ii;
        ii = i*i;
        ++ii;
        say(ii);
    }

    return 0;
}
```

#### 69.1.1 Invoking the debugger

After you compile your program (making sure to use the `-g` flag; see Tutorials book [11], section 2.2.4), instead of running it the normal way, you invoke `gdb`:

```
gdb myprogram
```

That puts you in an environment, recognizable by the `(gdb)` prompt:

```
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7
[stuff]
(gdb)
```

where you can do a controlled run of your program with the `run` command:

```
(gdb) run
Starting program: /home/eijkhout/gdb/hello
hello world 1
hello world 2
hello world 5
```

```
hello world 10
hello world 17
hello world 26
hello world 37
hello world 50
hello world 65
hello world 82
[Inferior 1 (process 30981) exited normally]
```

## 69.2 Example: integer overflow

The following program shows *integer overflow*. (We are using `short int` to force this to happen soon.)

| Code:                                                                                                                                                                                                                                                  | Output:                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>// gdb/overflow.cpp void say(short n) {     cout &lt;&lt; "hello world " &lt;&lt; n &lt;&lt; '\n'; }  int main() {      for (short i=0; ; i+=20) {         short ii;         ii = i*i;         ++i;         say(ii);     }      return 0; }</pre> | <pre>[debug] overflow:  hello world 0 hello world 441 hello world 1764 hello world 3969 hello world 7056 hello world 11025 hello world 15876 hello world 21609 hello world 28224 hello world -29815 hello world -21436 hello world -12175 hello world -2032 hello world 8993 hello world 20900 hello world -31847 hello world -18176 hello world -3623 hello world 11812 hello world 28129</pre> |

Let's do a walkthrough

```
(gdb) list
18     void say(short n) {
19         cout << format
20             ("hello world {}\\n", n);
(gdb) break 19 if n<0
Breakpoint 1 at 0x4044a5: file overflow.
   cpp, line 20.

(gdb) break 19 if n<0
Breakpoint 1 at 0x4044a5: file overflow.
   cpp, line 20.
(gdb) run
```

We see where the erroneous output is generated, so we put a conditional break there.

```
Starting program: overflow
hello world 0
[...]
hello world 28224
```

```
Breakpoint 1, say (n=-29815) at overflow.  
cpp:20  
20      ("hello world {}\\n", n);
```

```
(gdb) where  
#0  say (n=-29815) at overflow.cpp:20  
#1  0x000000000404546 in main () at  
    overflow.cpp:29  
(gdb) frame 1  
#1  0x000000000404546 in main () at  
    overflow.cpp:29  
29      say(ii);  
  
(gdb) print ii  
$1 = -29815  
(gdb) print i  
$2 = 190  
(gdb) print i * i
```

Running the programming takes us to the troublesome spot. a conditional break there.

Get the call stack and the calling frame.

Print the offending input, see where it comes from, and compute the result, which is indeed more than  $2^{15}$ .

## 69.3 More gdb

### 69.3.1 Run with commandline arguments

This program is self-contained, but if you had a program that takes *commandline arguments*:

```
./myprogram 25
```

you can supply those in gdb:

```
(gdb) run 25
```

### 69.3.2 Source listing and proper compilation

Inside gdb, you can get a source listing with the *list* command. You need to supply the *-g* compiler option for the *symbol table* to be included in the binary; see for yourself what happens when you leave out this option.

```
[] icpc -g -o hello hello.cpp  
[] gdb hello  
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7  
[stuff]  
Reading symbols from /home/eijkhout/gdb/hello...done.  
(gdb) list  
13      using std::cout;  
14      using std::endl;  
[et cetera]
```

(If you hit return now, the list command is repeated and you get the next block of lines. Doing *list -* gives you the block above where you currently are.)

### 69.3.3 Stepping through the source

Let's now make a more controlled run of the program. In the source, we see that line 22 is the first executable one:

```
20     int main() {
21
22     for (int i=0; i<10; i++) {
23         int ii;
24         ii = i*i;
...
}
```

We introduce a *breakpoint* with the *break* command:

```
(gdb) break 22
Breakpoint 1 at 0x400a03: file hello.cpp, line 22.
```

(If your program is spread over multiple files, you can specify the file name: `break otherfile.cpp:34`.)

Now if we run the program, it will stop at that line:

```
(gdb) run
Starting program: /home/eijkhout/gdb/hello

Breakpoint 1, main () at hello.cpp:22
22     for (int i=0; i<10; i++) {
```

To be precise: the program is stopped in the state before it executes this line.

We can now use *cont* (for ‘continue’) to let the program run on. Since there are no further breakpoints, the program will run to completion. This is not terribly useful, so let us change our minds about the location of the breakpoint: it would be more useful if the execution stopped at the start of every iteration.

Recall that the breakpoint had a number of 1, so we use *delete* to remove it, and we set a breakpoint inside the loop body instead, and continue until we hit it.

```
(gdb) delete 1
(gdb) break 23
Breakpoint 2 at 0x400a29: file hello.cpp, line 23.
(gdb) cont
Continuing.
Breakpoint 2, main () at hello.cpp:24
24     ii = i*i;
```

(Note that line 23 is not executable, so execution stops on the first line after that.)

Now if we continue, the program runs until the next break point:

```
(gdb) cont
Continuing.
hello world 5

Breakpoint 1, main () at hello.cpp:24
24     ii = i*i;
```

To get to the next statement, we use *next*:

```
(gdb) next  
25          ii++;  
(gdb)
```

Hitting return re-executes the previous command, so we go to the next line:

```
(gdb)  
26          say(ii);  
(gdb)  
hello world 2  
  
Breakpoint 1, main () at hello.cpp:24  
24          ii = i*i;
```

You observe that the function call

1. is executed, as is clear from the `hello world 1` output, but
2. is not displayed in detail in the debugger.

The conclusion is that `next` goes to the next executable statement in the current subprogram, not into functions and such that get called from it.

If you want to go into the function `say`, you need to use `step`:

```
(gdb) next  
25          ii++;  
(gdb) next  
26          say(ii);  
(gdb) step  
say (n=10) at hello.cpp:17  
17          cout << "hello world " << n << endl;
```

The debugger reports the function name, and the names and values of the arguments. Another ‘step’ executes the current line and brings us to the end of the function, and the next ‘step’ puts us back in the main program:

```
(gdb)  
hello world 10  
18          }  
(gdb)  
main () at hello.cpp:24  
24          ii = i*i;
```

#### 69.3.4 Inspecting values

When execution is stopped at a line (meaning right before it is executed) you can inspect any values in that subprogram:

```
24          ii = i*i;  
(gdb) print i  
$1 = 4
```

You can even let expressions be evaluated with local variables:

```
(gdb) print 2*i  
$2 = 8
```

## 69. Debugging with gdb

---

You can combine this looking at values with breakpoints. Say you want to know when the variable `ii` gets more than 40:

```
(gdb) break 26 if ii>40
Breakpoint 1 at 0x40009cd: file hello.cpp, line 26.
(gdb) run
Starting program: /home/eijkhout/intro-programming-private/code/gdb/hello
hello world 1
hello world 2
hello world 5
hello world 10
hello world 17
hello world 26
hello world 37

Breakpoint 1, main () at hello.cpp:26
26      say(ii);
Missing separate debuginfos, use: debuginfo-install glibc-2.17-292.el7.x86_64 libgcc
-4.8.5-39.el7.x86_64 libstdc++-4.8.5-39.el7.x86_64
(gdb) print i
$1 = 7
```

### 69.3.5 A NaN example

The following program:

```
17      float root(float n)
18      {
19          float r;
20          float n1 = n-1.1;
21          r = sqrt(n1);
22          return r;
23      }
24
25      int main() {
26          float x=9,y;
27          for (int i=0; i<20; i++) {
28              y = root(x);
29              cout << "root: " << y << endl
30              ;
31              x -= 1.1;
32          }
33          return 0;
34      }
```

prints some numbers that are ‘not-a-number’:

```
[] ./root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214
root: -nan
root: -nan
root: -nan
```

Suppose you want to figure out why this happens.

The line that prints the ‘nan’ is 29, so we want to set a breakpoint there, and preferably a conditional breakpoint. But how do you test on ‘nan’? This takes a little trick.

```
(gdb) break 29 if y!=y
Breakpoint 1 at 0x400ea6: file root.cpp, line 28.
(gdb) run
Starting program: /home/eijkhout/intro-programming-private/code/gdb/root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214

Breakpoint 1, main () at root.cpp:29
29          cout << "root: " << y << endl;
```

We discover what iteration this happens:

```
(gdb) print i
$1 = 8
```

so now we can rerun the program, and investigate that particular iteration:

```
(gdb) break 28 if i==8
Breakpoint 2 at 0x400eaf: file root.cpp, line 28.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/eijkhout/intro-programming-private/code/gdb/root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214

Breakpoint 2, main () at root.cpp:28
28          y = root(x);
```

We now go into the `root` routine to see what is going wrong there:

```
(gdb) step
root (n=0.200000554) at root.cpp:20
20          float n1 = n-1.1;
(gdb)
21          r = sqrt(n1);
(gdb) print n
$2 = 0.200000554
(gdb) print n1
$3 = -0.89999944
(gdb) next
22          return r;
(gdb) print r
$4 = -nan(0x400000)
```

And there we have the problem: our input  $n$  is used to compute another number  $n_1$  of which we compute the square root, and sometimes this number gets negative. In this simple example that's the end of the story, but in a realistic example you would now probably have discovered a flaw in the program logic, and you can fix your code.

### 69.3.6 Assertions

Instead of running a program and debugging it if you happen to spot a problem (and note that this may not always be the case!) you can also make your program more robust by including *assertions*. These are things that you know should be true, from your knowledge of the problem you are solving.

For instance, in the previous example there was a square root function, and you just ‘knew’ that the input was always going to be positive. So you edit your program as follows:

```
// header to allow assertions:  
#include <cassert>  
  
float root(float n)  
{  
    float r;  
    float n1 = n-1.1;  
    assert(n1>=0); // NOTE!  
    r = sqrt(n1);  
    return r;  
}
```

Now if you run your program, you get:

```
[] ./assert  
root: 2.81069  
root: 2.60768  
root: 2.38747  
root: 2.14476  
root: 1.87083  
root: 1.54919  
root: 1.14018  
root: 0.447214  
assert: assert.cpp:22: float root(float): Assertion `n1>=0' failed.  
Aborted (core dumped)
```

What does this give you?

- It only tells you that an assertion failed, not with what values;
- it does not give you a traceback or so; on the other hand
- assertions can help you detect error conditions that you might otherwise have overlooked!

## Chapter 70

### Complexity

#### 70.1 Theory

For the theory of complexity, see HPC book [13], section 20.

#### 70.2 Time complexity

**Exercise 70.1.** For each number  $n$  from 1 to 100, print the sum of all numbers 1 through  $n$ .

There are several possible solutions to this exercise. Let's assume you don't know the formula for the sum of the numbers  $1 \dots n$ . You can have a solution that keeps a running sum, and a solution with an inner loop.

**Exercise 70.2.** How many operations, as a function of  $n$ , are performed in these two solutions?

#### 70.3 Space complexity

**Exercise 70.3.** Read numbers that the user inputs; when the user inputs zero or negative, stop reading. Add up all the positive numbers and print their average.

This exercise can be solved by storing the numbers in a `std::vector`, but one can also keep a running sum and count.

**Exercise 70.4.** How much space do the two solutions require?



# Chapter 71

## Support tools

Knowing how to write a program is not enough: you need a variety of tools to be a good programmer, or to be a programmer at all.

### 71.1 Editors and development environments

Simple programs, such as most of the exercises in this book, can be written using a simple editor. Traditionally, programmers have used *emacs* or *vi* (or one of its derivates such as *vim*).

More sophisticated are development environments such as *Microsoft Visual Studio Code* or *CLion*.

### 71.2 Compilers

For the simple exercises in this book, all you needed to know about a compiler was the commandline

```
icpx -o myprogram myprogram.cpp
```

(or whatever compiler name and program name you had).

There is much more to know about compilers; see Tutorials book [11], chapter 2.

### 71.3 Build systems

For programs that are more complicated than a single source file (and even then...) you are wise to use some form of build system.

- The simplest and oldest solution is *Make*; see Tutorials book [11], chapter 3.
- More modern, more powerful, yet also in a way more complicated, is *CMake*; see Tutorials book [11], chapter 4.
- Make and Cmake are often integrated into the above-mentioned development environments.

### 71.4 Debuggers

If your code misbehaves, having a good debugger can be a lifesaver. The traditional debugger is *gdb* (see Tutorials book [11], chapter 11) but, again, this is often integrated into build environments.



## **PART VI**

### **INDEX AND SUCH**



#ifndef, 286

abstraction, 73

accessor, 109

actor model, 557

allocation

- automatic, 157
- dynamic, 157

Amazon

- delivery truck, 529
- Prime, 29
- prime, 529, 536

Apple, 23

argument, 77

- actual, 399
- default, 87
- dummy, 399
- keyword, 400
- optional, 401
- positional, 400

array, 135

- associative, 308
- assumed-shape, 432
- automatic, 425, 432
- initialization, 426
- operations, semantics of, 427
- rank, 428
- section, 427
- shape (Fortran), 431
- static, 425
- variable length, 162

ascii, 406

assembly

- coding, 537

assertion, 644

assignment, 42

asterisk

- in Fortran formatted I/O, 450

autotools, see GNU, autotools

bandwidth, 351

base, 44

binary I/O, see I/O, binary

bisection, 74, 487

BLAS, 589

BLIS, 589

Boost, 174

bottom-up, 591

Boyer-Moore, 560

breakpoint, 640

bubble sort, 166, 612

bug, 17

bus error, 138

C

- C11, 162
- C99, 51, 162, 306
- parameter passing, 250–254
- pointer, 245–255, 363
- preprocessor, 371
- string, 175, 362

C preprocessor, see preprocessor

C++, 354

- C++03, 355
- C++11, 173, 227, 266, 312, 356, 356
- C++14, 266, 356
- C++17, 44, 58, 122, 145, 173, 174, 224, 265, 302, 313, 315, 318, 330, 339, 348, 356, 357
- C++20, 38, 141, 145, 154, 161, 177, 178, 203, 207, 219, 224, 266, 272, 295, 303, 328, 330, 331, 335, 339, 348, 350, 351, 357, 361, 414, 598
- C++23, 38, 51, 131, 154, 155, 159, 177, 178, 181, 182, 196, 210–212, 261, 285, 317, 331, 357, 361, 540
- C++26, 155, 161

Core Guidelines, 593–594

- Enum.3, 330
- ES.50, 348
- R.1, 158, 194
- R.10, 158
- SF.5, 270
- SF.7, 279
- SF.8, 286

cache, 541

cache-oblivious programming, 537

Caesar cipher, 172

calendars, 357

call-back, 349

callable, 335

calling environment, 83, 229

capture, 197, 200

capture-default variable, 204  
case sensitive, 41  
cast, 49, 345  
    lexical, 174  
Catch2, 507, 628  
    installing, 633–634  
cellular automaton, 543, 543  
charconv, 174  
class, 99, 99  
    abstract, 116  
    base, 114  
    derived, 114  
    iteratable, 227  
    name injection, 291  
    pure virtual, 116  
CLion, 38, 647  
clock  
    resolution, 329  
closure, 94, 197  
CMake, 283, 583, 584, 588  
CMake (unix command), 37, 647  
Cmake, 634  
co-routine, 335  
code  
    duplication, 77  
    maintainance, 593  
code reuse, 77  
Collatz conjecture, 71  
column major, 160  
column-major, 156, 429, 450  
comma operator, 160  
command  
    history, 585  
commandline arguments, 50, 378  
compilation  
    separate, 269, 414, 514  
compile-time constant, 425  
compiler, 24, 35  
    and preprocessor, 281  
    one pass, 78  
compiling, 24  
complex numbers, 43, 305  
concept, 295, 295  
concepts, 357  
conditional, 53  
connected components, see graph, connected  
const  
    reference, 257  
constructor, 101, 243, 363  
    copy, 119, 258  
    for containers, 157  
    default, 101, 107  
    defaulted, 108, 109  
    delegating, 118, 482, 501  
    range, 361  
container, 307  
contains  
    for class functions, 419  
    in modules, 414  
continuation character, 373  
copy  
    deep, 424  
    shallow, 424  
copy constructor, see constructor, copy  
Core Guidelines, 30  
coroutines, 357  
Covid-19, 512  
cxxopts, 585–589  
data model, 350  
data race, 339, 340  
database, 335  
datatype, 40  
debugger, 647  
DEC  
    VT100, 359  
DEC PDP-11, 350  
declaration, 124  
    function, 78  
deducing this, see this, deducing  
#define, 283, 540  
definition, 124, 125  
definition vs use, 91  
dependence, 427  
dereference, 227, 246  
derefencing a  
    nullptr, 242  
destructor, 95, 120  
    at end of scope, 94  
    in Fortran, see final procedure  
Dijkstra  
    Edsger, 627

shortest path algorithm, 550  
directives, 281  
do  
    concurrent, 70, 438  
do loop  
    implicit, 450  
        and array initialization, 426  
    implied, 391  
dynamic  
    programming, 526

Ebola, 512  
Eclipse, 38  
ECMA, 311  
efficiency gap, 526  
Eigen, 151, 589  
eight queens, 316, 497  
#elifdef, 285  
#elifndef, 285  
emacs, 23, 23, 37, 373  
emacs (unix command), 647  
encoding  
    extendible, 175  
ENIAC, 567  
epoch, 328  
error  
    compile-time, 39  
    run-time, 39  
    syntax, 39  
#error, 286, 287  
exception, 138, 299  
    catching, 299  
    throwing, 299  
executable, 24, 36, 416  
execution  
    policy, 224  
execution policy, see range, execution policy  
exponent part, 44  
expression, 42  
extent, 160  
    of array dimension, 431

false sharing, 349  
Fasta, 559  
Fastq, 560  
field, 293  
file  
binary, 35, 39  
executable, 269  
handle, 120  
include, 99  
object, 269, 416  
source, 35  
final, 421  
floating point number, 44  
fmtlib, 177  
for  
    indexed, 140  
    range-based, 139  
Fortran, 156  
    90, 371  
    case ignores, 372  
    comments, 374  
    Fortran2003, 374, 426, 455, 459  
    Fortran2008, 416, 459  
    Fortran2018, 387, 391, 459  
    Fortran4, 459  
    Fortran66, 374, 459  
    Fortran77, 374, 459  
    Fortran88, 459  
    Fortran8X, 459  
    Fortran90, 387, 414, 459  
    Fortran95, 374, 377, 459  
forward declaration, 86  
    of classes, 93  
    of functions, 93  
friend, 117  
function, 73, 396, 396  
    argument, 76  
    arguments, 75  
    body, 76  
    call, 73, 76  
    declaration, 76  
    defines scope, 76  
    definition, 73, 75  
    header, 78  
    parameter, 76  
    parameters, 75  
    prototype, 78, 268  
    recursive, see also recursion  
    result type, 75  
    signature, 78  
function try block, 302

functional programming, 80, 229  
 functor, 128, 320  
**gdb**  
     break, 640  
     cont, 640  
     delete, 640  
     list, 639  
     next, 640  
     run, 637  
     run with commandline arguments, 639  
     step, 641  
**gdb** (unix command), 647  
**gerrymandering**, 521  
**Git**, 584  
**glyph**, 175  
**GNU**, 35  
     autotools, 584  
**Goldbach conjecture**, 472  
**Google**, 515  
     developer documentation style guide, 465  
     Pagerank, 515  
**graph**  
     connected, 517  
     diameter, 517  
     directed, 611  
     unweighted, 613  
**greedy search**, see search, greedy  
**has-a relation**, 111  
**hdf5**, 449  
**header**, 36, 40, 514  
     file, 282  
     guard, 286  
     vs modules, 357  
**header file**, 267, 269  
     and global variables, 272  
     treatment by preprocessor, 272  
**heap**, 151, 157  
     fragmentation, 157  
**hexadecimal**, 245  
**history**  
     command, 36  
**Holmes**  
     Sherlock, 172  
**homebrew**, 23  
**Horner's rule**, 293, 489  
**host association**, 404  
**Hypre**, 589  
**I/O**  
     binary, 189  
     formatted, 449  
     list-directed, 449  
     unformatted, 449  
**identifier**, 51  
**IEEE**  
     754, 45, 352  
**if**  
     ternary, 58  
**#if**, 285  
**#ifdef**, 285, 287  
**#ifndef**, 284, 285  
**include**  
     path, 282  
**#include**, 273, 281, 282  
**increment**  
     operator, 227  
**incubation period**, 512  
**inheritance**, 114  
**initialization**  
     variable, 40  
**initializer**  
     in conditional, 58  
     list, 114, 136, 139, 145  
     member, 104, 302, 482  
     statement, 141, 315  
**inline**, 401  
**integer**  
     fixed width types, 350  
     overflow, 638  
**interface**, 116  
**interrupt**, 557  
**invariant**, 110  
**is-a relation**, 114  
**ISO**  
     bindings, 380  
**iteration**  
     of a loop, see loop, iteration  
**iterator**, 207, 213, 213, 227, 356  
**keywords**, 39  
**kind selector**, 376  
**Kokkos**, 161

label, 452  
lambda, see closure, 349  
capture  
    by reference, 208  
expression, 197  
generic, 204  
immediately invoked, 197  
Lapack, 589  
lazy evaluation, 210  
lazy execution, 210  
lexicographic ordering, 65  
library  
    header-only, 292, 583  
    software, 583  
    standard, 40  
line printer, 455  
linear regression, 572  
linker, 269, 416  
Linux, 23  
list  
    linked, 601–608  
        in Fortran, 445–448  
    single-linked, 226  
locality, 427  
logging, 191  
loop, 61  
    body, 61  
    counter, 61  
    for, 61  
    header, 61  
    index, 62  
    inner, 65  
    invariant, 621  
    iteration, 61  
    nest, 65  
    outer, 65  
    range-based, 156  
    variable, 62  
    while, 61  
lvalue, 353  
  
macports, 23  
Make, 269, 283, 583  
Make (unix command), 37, 647  
makefile, 270, 514  
Manhattan distance, 529  
  
mantissa, 44  
Markov chain, 517  
math  
    functions (in C++), 87–88  
matrix  
    adjacency, 611  
member  
    data, 99  
    function, 99  
    initializer, see initializer, member, 149  
    initializer list, 129  
memoization, 526, 614  
memory  
    bottleneck, 541  
    leak, 120, 158, 238, 252, 252, 254, 433  
Mersenne twister, 322  
method, 105  
    overriding, 115  
methods, see member, function  
Microsoft  
    Visual Studio Code, 647  
    Windows, 23  
MKL, 589  
module, 99, 272  
    C++20, 357  
    sub, 416  
monadic extensions, 317  
move semantics, 354  
MPI, 335  
multicore, 351  
multiplication  
    Egyptian, 85  
Mumps, 589  
  
namespace, 275  
nano, 37  
newline, 39  
Newton's method, 492  
NP complete, 534  
NP-hard, 533  
null pointer, 242  
    from algorithms, 318  
    from dynamic cast, 347  
null terminator, 362  
  
object, 99  
    state of, 106

object file, see file, object  
opaque handle, 346  
OpenFrameworks, 355  
OpenMP, 335  
operator  
    arithmetic, 222  
    bitwise, 55  
    comparison, 54  
    logic, 54  
    overloading, 127, 484, 579  
        and copies, 355  
        of parentheses, 128  
    precedence, 55  
    spaceship, 357  
    unary star, 214  
opt2, 534  
output  
    binary, 455  
    raw, 455  
    unformatted, 455  
  
#pack, 286  
package manager, 23, 584  
parameter, 77, see also function, parameter  
    actual, 77  
    formal, 77, 400  
    input, 83  
    output, 83, 312  
    pass by value, 81  
    passing, 80  
        by reference, 229, 243, 363  
        by value, 229  
    passing by reference, 80, 83  
        in C, 83  
    passing by value, 80  
    throughput, 83  
parametrized, 411  
Pascal's triangle, 166  
pass by reference, see parameter, passing by reference  
pass by value, see parameter, passing by value  
path  
    Hamiltonian, 560  
perfect forwarding, 337  
PETSc, 303, 589  
pkgconfig, 635  
  
pointer, 129  
    arithmetic, 250  
    bare, 363, 609  
    decay, 163, 249  
    dereference, 214  
    derefencing, 441  
    null, see null pointer, 601  
    smart, 159, 235  
    unique, 240  
    void, 349  
    weak, 242, 244  
polymorphism, 127  
    of constructors, 109  
pop, 601  
POSIX, 326  
#pragma, 287  
#pragma once, 286  
precision  
    double, 293  
    single, 293  
preprocessor, 36, 281  
    and header files, 272  
    conditional, 285  
    conditionals, 285–286  
    macro  
        parameterized, 284  
        macros, 283–285  
procedure, 393  
    final, 421  
    internal, 401, 404  
    module, 401  
program  
    statements, 36  
programming  
    dynamic, 524  
    parallel, 70  
punch card, 371, 373  
push, 601  
putty, 23  
python, 21  
  
quantor, 218  
quicksort, 154, 612  
  
radix point, 44  
RAII, 362  
random

seed, 458  
random number  
  generator, 322, 326  
    Fortran, 458  
  seed, 327  
random walk, 595  
range  
  adapter, 209  
  adaptor, 209  
  execution policy, 223  
  view, 209  
range-based for loop, see for, range-based  
range-v3, 208  
ranges, 357  
recurrence, 439  
recursion, see function, recursive  
  depth, 88  
  mutual, 86  
reduction, 221  
  operator, 222  
  sum, 221  
reference, 82, 230  
  argument, 243, 363  
  const, 147, 230, 233  
    to class member, 230  
  to class member, 230  
reference count, 240  
regression test  
  seetesting, regression, 627  
regular expression, 309  
reserved words, 41  
return, 76  
  makes copy, 233  
root finding, 487  
row major, 160  
row-major, 156  
runtime error, 297  
rvalue, 353  
  reference, 354  
scope, 78, 91  
  and stack, 157  
  dynamic, 94  
  in conditional branches, 57  
  lexical, 91, 94  
  of function body, 76  
search  
  greedy, 532, 533  
section, see array, section  
segmentation fault, 138  
shell  
  inspect return code, 51  
short-circuit evaluation, 57, 218  
side-effects, 260  
significand, 44  
Single Source Shortest Path, 517  
Single Source Shortest Path (SSSP), 549  
SIR model, 510  
sleep (unix command), 335  
smatch, 310  
software library, see library, software  
source  
  format  
    fixed, 371  
    free, 371  
source code, 24  
span  
  subspan, 155  
stack, 88, 95, 151, 157, 432, 575, 601  
  array allocation on, 162  
  overflow, 88, 158, 432  
  pointer, 575  
Standard Template Library, 305  
state, 107  
statement functions, 401  
stream, 177, 188  
string, 170  
  concatenation, 170  
  null-terminated, 175  
  raw literal, 173, 311  
  size, 170  
  stream, 173  
structured binding, 308, 311, 314, 547, 550  
subprogram, see function  
sum, reduction, see reduction, sum  
superuser, 585  
symbol  
  table, 639  
syntax  
  error, 297  
system test  
  seetesting, system, 627

tab  
completion, 36  
`tar` (unix command), 584  
template, 495  
and separate compilation, 292  
argument deduction, 145  
parameter, 289  
Terminal, 23  
terminal  
emulator, 359, 544  
testing  
regression, 627  
system, 627  
unit, 627  
text  
formatting, 357  
this  
capture of, 203  
deducing, 261  
thread, 557  
time point, 328  
time zones, 357  
timer  
resolution, 458  
top-down, 591  
Traveling Salesman Problem (TSP),  
`textbf` 529  
tree, 609–611  
Trilinos, 589  
tuple, 312  
denotation, 313  
type  
deduction, 139  
derived, 409  
nullable, 315  
return  
trailing, 88  
undefined behavior, 278  
underscore, 51  
double, 51  
Unicode, 51, 175  
unit, 453  
unit test, see testing, unit  
Unix, 23  
use-after-free, 240  
UTF8, 175  
values  
boolean, 46  
variable, 40  
assignment, 40  
declaration, 40, 41, 41  
global, 41, 272  
in Fortran module, 575  
in header file, 272  
initialization, 42  
lifetime, 92  
numerical, 43  
shadowing, 92  
static, 94, 403  
vector, 275, 473  
bounds checking, 137  
methods, 142  
subvector, 217  
vi, 23, 23  
`vi` (unix command), 647  
view, see range, view  
vim, see vi, 37  
Virtualbox, 23  
Visual Studio, 23, 38  
VMware, 23  
VSCode, 38  
VT100  
cursor control, 359, 544  
#warning, 287  
web server, 335  
Windows, 350  
X windows, 23  
XCode, 38  
Xcode, 23

## Chapter 72

### Index of C++ keywords

\_Bool, 51  
\_\_FILE\_\_, 302  
\_\_LINE\_\_, 302  
\_\_STC\_NO\_VLA\_\_, 162  
\_\_cplusplus, 355  
  
abort, 298  
abs, 87, 306  
accumulate, 212, 219, 221, 223–225  
accumulate, 221  
addressof, 83, 233, 363  
adjacent, 212  
adjacent\_difference, 226  
adjacent\_find, 225  
algorithm, 26  
algorithm, 218  
all\_of, 225, 477  
all\_of, 218  
any, 321  
any\_cast, 321  
any\_of, 205, 218, 219, 225, 477  
any\_of, 218  
argc, 378, 586  
argv, 586  
array, 145, 307  
array, 144  
as\_const, 348  
assert, 631  
assert, 298  
async, 338  
at, 137, 138, 142  
auto, 139, 356  
auto\_ptr, 355, 356  
  
back, 142, 214  
  
bad\_alloc, 302  
bad\_alloc, 157  
bad\_cast, 347  
bad\_exception, 302  
bad\_optional\_access, 316  
bad\_variant\_access, 319  
basic\_ios, 194  
begin, 207, 213  
begin, 213  
bfloat16\_t, 45  
binary\_search, 225  
bit\_cast, 348  
bitset, 186  
bool, 188, 352  
boolalpha, 188  
break, 66  
  
cartesian\_product, 160  
catch, 299  
cerr, 191, 361  
char, 350  
char, 169  
cin, 192, 361  
cin, 47  
clamp, 225  
close, 188  
cmath, 47  
cmp\_equal, 351  
cmp\_greater, 351  
cmp\_not\_equal, 351  
complex, 352  
complex, 305  
conj, 306  
const, 257, 259, 265  
const\_cast, 264, 348

consteval, 357  
constexpr, 265, 356  
constexpr, 265  
constinit, 357  
continue, 68  
copy, 225  
copy, 215  
copy\_backward, 225  
copy\_if, 226  
copy\_n, 225  
count, 225  
count\_if, 225  
cout, 40, 46, 361  
cxxopts, 477  
data, 155  
decltype, 345  
default\_random\_engine, 322  
delete, 158  
denorm\_min, 307  
destroy, 225  
destroy\_at, 225  
destroy\_n, 225  
distance, 216  
divides, 222  
double, 45  
duration\_cast, 329  
duration\_count, 328  
dynamic\_cast, 347  
emplace, 317  
emplace\_back, 337  
end, 207, 213, 214  
end, 213  
endl, 191  
enum, 330  
enum class, 330  
enum struct, 330  
EOF, 193  
eof, 193  
epsilon, 307  
equal, 225  
equal\_range, 225  
erase, 143, 144  
erase, 215  
exception, 300  
exclusive\_scan, 224, 226  
ExecutionPolicy, 224  
exit, 50  
EXIT\_FAILURE, 50  
EXIT\_SUCCESS, 50  
exp, 306  
expected, 317  
export, 273  
export, 273  
extent, 160  
fabs, 87  
false, 46, 48  
FILE, 194  
filesystem, 330  
fill, 225  
fill\_n, 225  
filter, 477  
filter, 210  
find, 216, 225, 309  
find\_end, 225  
find\_first\_of, 225  
find\_if, 225, 309, 551, 552  
find\_if\_not, 225  
first, 312  
fixed, 186  
float, 45  
float16\_t, 45  
flush, 191  
for, 138, 140  
for\_each, 208, 218, 221, 225  
for\_each, 220  
for\_each\_n, 225  
format, 178  
formatter, 181, 195  
free, 158  
friend, 191  
from\_chars, 174  
front, 142  
function, 199  
future, 337  
gcd, 225  
generate, 225  
generate\_n, 225  
get, 240, 312, 318, 319, 337  
get\_if, 318  
get\_if, 318

getline, 192, 193  
getline, 192  
getrusage, 330  
  
has\_quiet\_NaN, 352  
has\_value, 315  
high\_resolution\_clock, 328  
hours, 328  
  
if, 58  
ifstream, 194  
import, 273  
import, 273  
includes, 225  
inclusive\_scan, 224, 226  
index, 319  
index, 319  
inline, 122  
inner\_product, 225  
inplace\_merge, 226  
insert, 143  
insert, 216  
int, 350, 352  
int, 44  
int16\_t, 350  
iota, 213, 225, 326, 477  
iota\_view, 210  
is\_eof, 194  
is\_heap, 225  
is\_heap\_until, 225  
is\_open, 194  
is\_partitioned, 225  
is\_permutation, 225  
is\_sorted, 225  
is\_sorted\_until, 225  
isfinite, 352  
isinf, 352  
isnan, 302, 352  
isnormal, 352  
iter\_swap, 225  
iterator, 213  
itoa, 174  
  
join, 335, 339  
jthread, 339  
  
knuth\_b, 322  
  
lcm, 225  
lexicographical\_compare, 225  
lock, 242  
lock\_guard, 339  
logical\_and, 222  
logical\_or, 222  
long, 350, 351  
long, 350  
long double, 45  
long int, 350, 351  
long long, 350  
long long int, 350  
lower\_bound, 225  
lowest, 307  
  
main, 50, 586  
make\_heap, 225  
make\_pair, 312  
make\_tuple, 311  
malloc, 158, 243, 253, 254, 361, 363  
malloc, 249  
map, 473, 550  
map, 308  
max, 87, 225  
max, 307  
max\_element, 216, 221, 225  
max\_element, 222  
MAX\_INT, 306  
mdspan, 154, 156, 159, 160, 357, 540, 553  
mdspan, 155  
mdsubspan, 161  
memcpy, 157  
memory\_buffer, 182, 195  
merge, 226  
microseconds, 328  
millisecond, 328  
milliseconds, 328  
min, 225  
min, 307  
min\_element, 221, 225  
MIN\_INT, 306  
minmax, 225  
minmax\_element, 225  
minus, 222  
minutes, 328  
mismatch, 225

module, 273  
modulus, 222  
monostate, 320  
move, 225  
move\_backward, 225  
mt19937, 322  
multiplies, 222  
mutable, 132, 348  
mutable, 204, 264  
  
namespace, 275  
nanoseconds, 328  
NDEBUG, 298  
negate, 222  
new, 149, 150, 158, 244, 251, 253, 254  
next\_permutation, 225  
noexcept, 302  
none\_of, 225  
none\_of, 218  
norm, 306  
now, 329  
nth\_element, 226  
NULL, 157, 175, 238, 242, 601  
nullopt, 315  
nullptr, 157, 237, 238, 242, 601  
nullptr\_t, 242  
numeric, 221  
numeric\_limits, 306  
  
ofstream, 188  
open, 188, 189, 193  
optional, 315, 317  
out\_of\_range, 300  
override, 115, 116  
  
pair, 312  
pair, 312  
partial\_ordering, 332  
partial\_sort, 226  
partial\_sort\_copy, 226  
partial\_sum, 224, 225  
partition, 226  
partition\_copy, 226  
partition\_point, 225  
period, 329  
plus, 222  
polar, 306  
  
pop\_back, 143  
pop\_heap, 225  
prev\_permutation, 225  
print, 178  
printf, 51, 177, 361  
println, 178  
private, 102, 105, 109, 110  
protected, 114  
public, 102, 109, 110  
push\_back, 143, 144  
push\_heap, 225  
puts, 177  
  
quiet\_NaN, 352  
  
RAII, 158  
rand, 326  
rand48, 326  
rand48\_r, 326  
RAND\_MAX, 326  
rand\_r, 326  
random, 326  
random\_device, 322  
random\_r, 326  
rbegin, 215, 293  
reduce, 223, 225  
reduce, 224  
regex, 309  
regex\_match, 309, 310  
regex\_search, 310  
reinterpret\_cast, 348  
remove, 226  
remove\_copy, 226  
remove\_copy\_if, 226  
remove\_if, 226  
rend, 215, 293  
replace, 225  
replace\_copy, 225  
replace\_copy\_if, 225  
replace\_if, 225  
reserve, 144  
return, 50  
reverse, 225  
reverse\_copy, 225  
rotate, 226  
  
sample, 225

scanf, 361  
scientific, 187  
scoped\_lock, 339  
search, 225  
search\_n, 225  
second, 312  
seconds, 328  
seconds, 327  
seed, 322  
set, 550  
set, 309  
set\_difference, 226  
set\_intersection, 226  
set\_symmetric\_difference, 226  
set\_union, 226  
setprecision, 186, 187  
setw, 183, 187  
shared\_ptr, 363, 602  
short, 351  
short, 350  
short int, 350, 351  
shuffle, 225, 326  
signalling\_NaN, 352  
size, 142, 155, 161  
size\_t, 157, 345  
sizeof, 164, 249, 253, 350  
sizeof, 249  
sleep, 330  
sleep\_for, 338  
sort, 226  
sort, 223  
sort\_heap, 225  
span, 155, 217, 357, 361, 539  
span, 154  
sprintf, 174  
srand, 327  
ssize, 157, 161  
stable\_partition, 226  
stable\_sort, 226  
static, 122, 324, 402  
static, 121  
static\_assert, 298  
static\_cast, 348, 495  
stderr, 191  
steady\_clock, 328  
string, 362  
string\_literals, 174  
string\_view, 173  
stringstream, 174, 191  
strong\_ordering, 332  
struct, 312  
swap, 354  
swap\_ranges, 225  
switch, 58  
switch, 56  
system\_clock, 328  
take, 213  
this, 130, 131, 203, 241, 260  
this, 129  
throw, 299  
to, 210  
to\_array, 145  
to\_chars, 174  
to\_string, 174  
to\_underlying, 331  
transform, 212, 221, 225  
transform, 210, 221  
transform\_exclusive\_scan, 224, 226  
transform\_inclusive\_scan, 224, 226  
transform\_reduce, 224, 226  
true, 46, 48  
tuple, 311  
tuple, 312  
typedef, 285  
typedef, 285  
uint16\_t, 350  
unexpected, 317  
uninitialized\_\*, 225  
union, 318  
unique, 226  
unique\_copy, 226  
unique\_ptr, 240, 363, 602, 608  
unordered\_map, 308  
upperbound, 225  
using, 285  
using namespace, 271  
utc\_clock, 328  
value, 315, 316  
value\_or, 316  
variant, 631, 633

variant, **318**  
vector, **135**, **139**, **158**, **209**, **217**, **290**, **307**, **361**,  
    **601**  
vector, **142**  
views, **209**  
virtual, **115**  
visit, **320**  
void, **76**  
void, **79**  
  
weak\_ptr, **242**  
while, **142**  
while, **68**  
  
zip, **211**  
zip\_transform, **212**  
zip\_with, **212**

## Chapter 73

### Index of Fortran keywords

.AND., 386  
.and., 386  
.eq., 386  
.false., 387  
.ge., 386  
.gt., 386  
.le., 386  
.lt., 386  
.ne., 386  
.or., 386  
.true., 387  
  
Abs, 434  
advance, 448  
aimag, 376  
AIMIG, 382  
All, 435  
all, 436  
allocatable, 374  
allocate, 432, 433, 445, 448  
Any, 435  
any, 436  
Associated, 444  
  
bit\_size, 379  
btest, 377  
  
c\_sizeof, 379  
call, 394, 397  
case, 386  
Char, 406  
Character, 375, 405  
CMPLX, 376, 382  
command\_argument\_count, 378  
Common, 404  
  
common, 414, 459  
Complex, 375, 377  
concurrent, 438  
CONJG, 382  
Contains, 393, 404  
contains, 397, 401, 414, 419, 420, 457  
continue, 391  
Count, 435  
Cshift, 434  
  
DBLE, 382  
deallocate, 433  
DIM, 435  
dimension, 374, 425  
dimension(:), 432  
do, 389–391, 403, 459  
DOT\_PRODUCT, 436  
Dot\_Product, 434  
  
end do, 390, 403  
End Program, 373  
entry, 401  
external, 457  
  
FLOAT, 382  
for all, 438  
forall, 437  
Function, 395, 433  
function, 396  
  
get\_command, 378  
get\_command\_argument, 378  
goto, 391, 459  
  
huge, 380

Iachar, 406  
iand, 377  
ibclr, 377  
ibits, 377  
ibset, 377  
Ichar, 407  
ieor, 377  
if, 385, 387, 459  
if, arithmetic, 387  
implicit none, 413, 414, 459  
in, 399  
inout, 399  
INT, 382  
Integer, 375  
integer, 374  
intent, 374  
Interface, 393  
interface, 397, 457, 457  
intrinsic, 401  
ior, 377  
iso\_c\_binding, 379  
kind, 379, 411  
Lbound, 431  
lbound, 426  
len, 405, 411  
Logical, 375, 377, 386  
logical, 374  
MASK, 435  
MATMUL, 436  
MatMul, 434  
MaxLoc, 434  
MaxVal, 435  
MinLoc, 434  
MinVal, 435  
Module, 393, 404  
module, 413  
mvbits, 377  
NINT, 382  
Nullify, 444  
o, 391  
Optional, 401  
optional, 401  
out, 399  
parameter, 374, 375, 375  
pointer, 441  
precision, 378  
Present, 401  
Print, 453  
private, 416, 422  
procedure, 420  
Product, 435  
Program, 372, 373, 393  
protected, 416  
public, 416  
random\_number, 458  
random\_seed, 458  
range, 378  
Read, 454  
REAL, 382  
Real, 375  
real, 376  
real(4), 374  
real(8), 374, 380  
Recursive, 395  
recursive, 395  
RESHAPE, 431  
reshape, 429  
Result, 398, 433  
result, 396  
return, 394, 396  
Save, 404  
save, 401, 402  
Select, 387  
select, 386  
selected\_int\_kind, 378  
selected\_real\_kind, 378  
shape, 431  
size, 431  
SNGL, 382  
SPREAD, 431  
stat=iererror, 433  
stop, 372, 372  
storage\_size, 378, 379  
Subroutine, 433  
subroutine, 396  
Sum, 435

system\_clock, 458

target, 442

Transpose, 434

trim, 406

Type, 375, 420

type, 409, 419, 422

Ubound, 431

ubound, 426

use, 397, 413, 414, 422

variable

length of name, 374

where, 436

while, 390

Write, 382, 448, 453, 454



## Chapter 74

### Bibliography

- [1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings Supercomputing '90*, pages 2–11. IEEE Computer Society Press, Los Alamitos, California, 1990. [538](#)
- [2] Roy M. Anderson and Robert M. May. Population biology of infectious diseases: Part I. *Nature*, 280:361–367, 1979. doi:10.1038/280361a0. [506](#)
- [3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997. [349](#)
- [4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc webpage. <http://www.mcs.anl.gov/petsc>, 2011. [349](#)
- [5] Tolga Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006. [534](#)
- [6] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977. [560](#)
- [7] Yen-Lin Chen, Chuan-Yen Chiang, Yo-Ping Huang, and Shyan-Ming Yuan. A project-based curriculum for teaching C++ object-oriented programming. In *Proceedings of the 2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing, UIC-ATC '12*, pages 667–672, Washington, DC, USA, 2012. IEEE Computer Society. [29](#)
- [8] C++ core guidelines. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>. [158](#), [194](#), [270](#), [279](#), [286](#), [330](#), [348](#)
- [9] <https://github.com/jarro2783/cxxopts>. [544](#)
- [10] David Donoho, Mahsa Lotfi, and Batu Ozturkler. The mathematics of mass testing for covid-19, 2020. <https://www.siam.org/publications/siam-news/articles/the-mathematics-of-mass-testing-for-covid-19>. [513](#)
- [11] Victor Eijkhout. HPC carpentry. <https://theartofhpc.com/carpentry.html>. [270](#), [283](#), [465](#), [584](#), [585](#), [637](#), [647](#)
- [12] Victor Eijkhout. Parallel programming in mpi and openmp. <http://theartofhpc.com/pcse.html>. [224](#), [536](#)
- [13] Victor Eijkhout. The science of computing. <http://theartofhpc.com/istc.html>. [20](#), [45](#), [309](#), [350](#), [351](#), [492](#), [517](#), [518](#), [534](#), [537](#), [538](#), [541](#), [549](#), [550](#), [551](#), [553](#), [611](#), [613](#), [645](#)

- 
- [14] S.C. Eisenstat and H.F. Walker. Choosing the forcing terms in an inexact Newton method. *SIAM J. Sci. Comput.*, 17:16–32, 1996. [494](#)
  - [15] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Calling 17, page 20?29, New York, NY, USA, 2017. Association for Computing Machinery. [29](#)
  - [16] Google. Google developer documentation style guide. <https://developers.google.com/style/>. [465](#)
  - [17] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):1–25, 2008. [537](#), [542](#)
  - [18] David S. Hollman, Bryce Adelstein Lelbach, H. Carter Edwards, Mark Hoemmen, Daniel Sunderland, and Christian R. Trott. mspan in c++: A case study in the integration of performance portable features into international language standards. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, volume 74, pages 60–70. IEEE, November 2019. [155](#)
  - [19] L.V. Kantorovich and G.P. Akilov. *Functional Analysis in Normed Spaces*. Pergamon press, 1964. [494](#)
  - [20] <https://mathworld.wolfram.com/Kermack-McKendrickModel.html>. [506](#)
  - [21] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. [538](#)
  - [22] <https://redistrictingonline.org/>. [521](#)
  - [23] Harry L. Reed. Firing table computations on the eniac. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, ACM ’52, page 103?106, New York, NY, USA, 1952. Association for Computing Machinery. [567](#)
  - [24] Juan M. Restrepo, , and Michael E. Mann. This is how climate is always changing. *APS Physics, GPS newsletter*, February 2018. [571](#)
  - [25] Lin S. and Kernighan B. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973. [534](#)
  - [26] [https://en.wikipedia.org/wiki/Epidemic\\_model](https://en.wikipedia.org/wiki/Epidemic_model). [510](#)
  - [27] [urlhttps://vt100.net/docs/vt100-ug/chapter3.html](https://vt100.net/docs/vt100-ug/chapter3.html). [359](#)