

2023

THE BOOK OF MODERN C++

Learn C++20/23 and other advanced topics from multiple experts.



里缪 404 何荣华 邹启翔 水滴会飞 整理

Not For Sale

The Book of Modern C++

Learn C++20/23 and other advanced topics from multiple experts.

里缪 404 何荣华 邹启翔 水滴会飞 整理

©2023 本书完全免费，所有文章版权归原作者所有，禁止以电子、机械、复印或其他任何形式出售！

发布日期：2023 年 5 月 18 日

发布地址：<https://github.com/lkimuk/the-book-of-modern-cpp>

邮件：lkimuk@outlook.com

封面：Midjourney

©2023 This book is completely free. All articles are copyrighted by the original authors, and selling in any form, including electronic, mechanical, photocopying, or otherwise, is prohibited!

First published: May 18, 2023

Published at <https://github.com/lkimuk/the-book-of-modern-cpp>

Email: lkimuk@outlook.com

Cover: Midjourney

This page is intentionally left blank.

Brief Contents

序	I
写作结构	II
难度等级	III
错误反馈	IV
I Basics	1
1 洞悉函数重载决议	2
2 Move semantics and rvalue references: Modern C++ fundamentals	43
3 To Save C, We Must Save ABI	52
II Modern C++	78
4 The Evolution of Functions in Modern C++	79
5 Design and evolution of constexpr in C++	94
6 C++20's parenthesized aggregate initialization has some downsides	123
7 (Non) Static Data Members Initialization, from C++11 till C++20	131
8 [C++] Static, Dynamic Polymorphism, CRTP and C++20's Concepts	142
9 Structured bindings in C++17, 5 years later	147
10 std::optional and non-POD C++ types	153
11 C++20 concepts are structural: What, why, and how to change it?	159
12 Most C++ constructors should be explicit	168
III C++ 23	175
13 Overview of C++23 Features	176
14 Three C++23 features for common use	211
15 Three new utility functions in C++23	216
16 C++23's Deducing this: what it is, why it is, how to use it	220
17 Using the C++23 std::expected type	230
18 C++23: Consteval if to make compile time programming easier	238
19 C++23: Narrowing contextual conversions to bool	242

20 6 C++23 features improving string and string_view	246
21 C++23: Preprocessing directives	250
22 C++23: The stacktrace library	252
23 C++23: flat_map, flat_set, et al.	257
24 C++23: How lambdas are going to change?	261
25 C++23: attributes	264
26 A Gentle Introduction to mdspan	267
27 C++23's new function syntax	272
28 Functional exception-less error handling with C++23's optional and expected	275
29 The evolution of enums	282
IV Metaprogramming	289
30 C++ 反射 第一章 通识	290
31 C++ 反射 第二章 探索	296
32 C++ 反射 第三章 原生	308
33 C++ 反射 第四章 标准	320
V Concurrency	334
34 STRUCTURED CONCURRENCY IN C++	335
35 Comparing Rust's and C++'s Concurrency Library	347
36 C++ 20 concurrency	353
37 co_resource<T>: An RAII coroutine	366
38 C++ Coroutines: Understanding the Compiler Transform	378
39 Back Porting C++20 Coroutines to C++14 – A Tutorial	417
40 Debugging C++ Coroutines	428
41 C++20 Coroutines and io_uring	444
42 Instruction-level parallelism in practice: speeding up memory-bound programs with low ILP	461
VI Performance	474
43 5 Ways to Initialize a String Member	475
44 Using final in C++ to improve performance	482
45 Technique: Compile Time Code Generation and Optimization	495
46 Improving the State of Debug Performance in C++	503
47 Simplify Code with if constexpr and Concepts in C++17/C++20	517

48 The memory subsystem from the viewpoint of software: how memory subsystem affects software performance	531
49 the sad state of debug performance in c++	557
50 Using Requires Expression in C++20 as a Standalone Feature	563
51 What is faster: vec.emplace_back(x) or vec[x] ?	571
52 P2723R0 Zero-initialize objects of automatic storage duration	576
53 wordexpr: compile-time wordle in c++20	587
54 const vs constexpr vs consteval vs constinit in C++20	591
VII Algorithms	597
55 C++20 Ranges Algorithms	598
56 New Fold Algorithms	634
57 Faster integer formatting - James Anhalt (jeaiii)' s algorithm	640
VIII Techniques	658
58 Conditional Members	659
59 Inside boost::unordered_flat_map	669
60 Static B-Trees	679
61 Technique: Recursive variants and boxes	697
62 Working with Strings in Embedded C++	704
63 Determining if a template specialization exists	728
64 How to Store an lvalue or an rvalue in the Same Object	737
65 Automatic Serialization in C++ for Game Engines	745
66 Technique: Take a constexpr string from Compile Time to Run Time	759
IX Tricks	765
67 如何优雅地打印类型名称?	766
68 Avoiding direct syscall instructions by using trampolines	774
69 Avoiding Temporaries with Expression Templates	778
70 Casting a negative float to an unsigned int	786
71 Compiler Tricks to Avoid ABI-Induced Crashes	790
72 Five tricky topics for data members in C++20	794
73 On finding the average of two unsigned integers without overflow	801

X Miscellaneous	811
74 New integer types I'd like to see	812
75 proxy: Runtime Polymorphism Made Easier Than Ever	816
76 C23 is Finished: Here is What is on the Menu	828
77 Cpp2 and cppfront: Year-end mini-update	847
78 Copy-Paste Developments	851
79 Carbon's most exciting feature is its calling convention	854
80 Ways to Refactor Toggle/Boolean Parameters in C++	862
81 malloc() and free() are a bad API	871
82 Use compiler flags for stack protection in GCC and Clang	878
83 The Wonderfully Terrible World of C and C++ Encoding APIs (with Some Rust)	882
The Links of All Articles	913

Contents

序	I
写作结构	II
难度等级	III
错误反馈	IV
I Basics	1
1 洞悉函数重载决议	2
1.1 重载决议的基本流程	2
1.2 Name Lookup	3
1.2.1 Qualified Name Lookup	3
1.2.2 Unqualified Name Lookup	7
1.2.3 Template Name Lookup	15
1.2.4 Two-phase Name Lookup	19
1.3 Function Templates Handling	22
1.3.1 Template Argument Deduction	22
1.3.2 Template Argument Substitution	26
1.4 Overload Resolution	29
1.4.1 Candidate functions	29
1.4.2 Viable functions	30
1.4.3 Tiebreakers	30
1.5 走一遍完整的流程	34
1.6 Name Mangling	37
1.7 总结	40
2 Move semantics and rvalue references: Modern C++ fundamentals	43
2.1 The value categories	43
2.1.1 lvalue vs prvalue	43
2.1.2 xvalue	44
2.1.3 Can be “moved from”	44
2.1.4 When to move-cast	45
2.1.5 Values types summary	46
2.2 Rvalue references	46

2.3	Taking advantage of move semantics	48
2.4	Type of this	51
3	To Save C, We Must Save ABI	52
3.1	The Monster - Application Binary Interface	53
3.2	The C ABI	54
3.3	ABI from the Other Side	56
3.4	Causing Problems (On Purpose)	57
3.5	Causing Problems (By Accident)	59
3.6	And That's the Good Ending	60
3.7	But...ABI?	61
3.8	An Old Solution	61
3.9	Transparent Aliases	63
3.10	Zero-Cost (Seriously, We Mean It™ This Time)	63
3.11	An Indirection Layer	65
3.12	The True Goal	67
3.13	The ABI Test: maxabs	67
3.14	Making a Shared Library	68
3.15	The “Applications”	71
3.16	No Free Lunch	73
3.17	A Great Success	76
3.18	I will embrace them.	76
II	Modern C++	78
4	The Evolution of Functions in Modern C++	79
4.1	Pre-C++11	79
4.2	C++11	81
4.2.1	Variadic function templates	81
4.2.2	Alternative function syntax	82
4.2.3	constexpr functions	83
4.2.4	Override and final specifiers for virtual functions	83
4.2.5	More special member functions	84
4.2.6	Default and deleted functions	86
4.2.7	Lambdas	87
4.3	C++14	88
4.3.1	Function return type deduction	88
4.3.2	Generic lambdas	88
4.4	C++20	89
4.4.1	Immediate functions	89
4.4.2	Abbreviated function templates	90
4.4.3	Lambda templates	91
4.4.4	constexpr virtuals	91

4.4.5	Coroutines	92
5	Design and evolution of constexpr in C++	94
5.1	C++98 and C++03: Ranks among const variables	95
5.2	0-∞: Constant evaluator in compiler	98
5.2.1	[SPOILER BLOCK BEGINS]	98
5.2.2	[SPOILER BLOCK ENDS]	99
5.3	2003: No need for macros	100
5.3.1	[SPOILER BLOCK BEGINS]	100
5.3.2	[SPOILER BLOCK ENDS]	100
5.4	2006-2007: When it all becomes clear	102
5.5	2007: First constexpr for data structures	104
5.6	2008: Recursive constexpr methods	104
5.7	2010: "const T&" as arguments in constexpr methods	104
5.8	2011: static_assert in constexpr methods	106
5.9	2012: (Almost) any code in constexpr functions	106
5.10	2013: (Almost) any code allowed in constexpr functions ver 2.0 Mutable Edition	107
5.11	2013: Legendary const methods and popular constexpr methods	109
5.12	2015-2016: Syntactic sugar for templates	110
5.13	2015: Constexpr lambdas	111
5.13.1	[SPOILER BLOCK BEGINS]	111
5.13.2	[[SPOILER BLOCK ENDS]]	112
5.14	2017-2019: Double standards	112
5.15	2017-2019: We need to go deeper	113
5.16	2017: The evil twin of the standard library	113
5.17	2017-2019: Constexpr gains memory	114
5.18	2018: Catch me if you can	116
5.19	2018: I said constexpr!	116
5.20	2018: Too radical constexpr	117
5.21	2020: Long-lasting constexpr memory	117
5.22	2021: Constexpr classes	120
5.23	2019-∞: Constant interpreter in the compiler	121
5.24	What else to look?	121
6	C++20's parenthesized aggregate initialization has some downsides	123
6.1	The basics of brace-initialization versus parens-initialization	123
6.2	Aggregate versus non-aggregate initialization	124
6.3	The coincidence of zero-argument initialization	127
6.4	emplace_back presents a stumbling block	127
6.5	C++20's solution: Parens-init for aggregates	128
6.6	The unintended consequences	128
6.7	The bottom line	130
7	(Non) Static Data Members Initialization, from C++11 till C++20	131

7.1	Initialisation of Data members	131
7.2	NSDMI - Non-static data member initialization	132
7.3	How It works	132
7.4	Copy and Move Constructors	134
7.5	Other forms of NSDMI	135
7.6	C++14 Updates for aggregates, NSDMI	136
7.7	C++20 Updates for bit fields	136
7.8	The case with auto	137
7.9	The case with CTAD - Class Template Argument Deduction	137
7.10	Advantages of NSDMI	138
7.11	Any negative sides of NSDMI?	138
7.12	Inline Variables C++17	138
7.13	Summary	139
7.14	Extra Links	140
8	[C++] Static, Dynamic Polymorphism, CRTP and C++20's Concepts	142
8.1	Dynamic Polymorphism	142
8.2	Static Polymorphism	143
8.3	C++20's Concepts	143
8.4	Conclusion	146
9	Structured bindings in C++17, 5 years later	147
9.1	The basics of structured binding	147
9.2	The Syntax	147
9.3	Modifiers	148
9.4	Binding	148
9.5	C++17/C++20 changes	149
9.6	Iterating through maps	150
9.7	Working with structures and arrays	151
9.8	Real code	151
10	std::optional and non-POD C++ types	153
10.1	The good	154
10.2	The ugly	155
10.3	The bad	156
10.4	Recommendation	157
10.5	Conclusion	158
11	C++20 concepts are structural: What, why, and how to change it?	159
11.1	Nominal vs. structural concepts	159
11.2	Nominal is better...	160
11.3	but structural is what C++ needs	162
11.4	Nominal concepts in C++20	163
11.5	Reverse nominal concepts	167

11.6	Conclusion	167
12	Most C++ constructors should be explicit	168
12.1	C++ gets the defaults wrong	169
12.2	Implicit is correct for copy and move constructors	170
12.3	Implicit is correct for types that behave like bags of data members	170
12.4	Implicit is correct for containers and sequences	172
12.5	Implicit is correct for string and function	173
12.6	Explicit is correct for most everything else	173
12.6.1	explicit operator bool() const	174
12.7	A stab at a complete guideline	174
III	C++ 23	175
13	Overview of C++23 Features	176
13.1	Deducing this (P0847)	176
13.2	Monadic std::optional (P0798R8)	180
13.3	std::expected (P0323)	181
13.4	Multidimensional Arrays (P2128)	182
13.5	if consteval (P1938)	183
13.6	Formatted Output (P2093)	184
13.7	Formatting Ranges (P2286)	185
13.8	import std (P2465)	188
13.9	out_ptr (P1132r8)	189
13.10	auto(x) decay copy (P0849)	190
13.11	Narrowing contextual conversions to bool (P1401R5)	192
13.12	forward_like (P2445)	192
13.13	#elifdef and #elifndef (P2334)	193
13.14	#warning (P2437)	193
13.15	constexpr std::unique_ptr (P2273R3)	193
13.16	Improving string and string_view (P1679R3, P2166R1, P1989R2, P1072R10, P2251R1)	193
13.17	static operator() (P1169R4)	196
13.18	std::unreachable (P0627R6)	196
13.19	std::to_underlying (P1682R3)	197
13.20	std::byteswap (P1272R4)	197
13.21	std::stacktrace (P0881R7, P2301R1)	198
13.22	Attributes (P1774R8, P2173R1, P2156R1)	199
13.23	Lambdas (P1102R2, P2036R3, P2173R1)	201
13.24	Literal suffixes for (signed) size_t (P0330R8)	202
13.25	std::mdspan (P0009r18)	203
13.26	flat_map, flat_set (P0429R9, P1222R4)	204
13.27	ranges::to (P1206R7)	204
13.28	Ranges: starts_with and ends_with (P1659)	205

13.29 Views: zip, zip_transform, adjacent, and adjacent_transform (P2321R2)	206
13.30 Ranges: fold (P2322R6)	207
13.31 Ranges: iota, shift_left, and shift_right (P2440R1)	207
13.32 Range adaptors: slide, chunk, and chunk_by (P2442R1, P2443R1)	208
13.33 Range adaptor: views::join_with (P2441R2)	209
13.34 std::generator (P2502R2)	209
13.35 Fix istream_view (P2432R1)	209
13.36 ranges::contains (P2302R4)	210
14 Three C++23 features for common use	211
14.1 Literal suffixes for size_t and ptrdiff_t	211
14.2 Multidimensional subscript operator	212
14.3 contains() member function for string/string_view	214
15 Three new utility functions in C++23	216
15.1 std::unreachable	216
15.2 std::to_underlying	217
15.3 std::byteswap	218
16 C++23's Deducing this: what it is, why it is, how to use it	220
16.1 Overview	221
16.2 Design	222
16.3 Use Cases	223
16.3.1 De-duplication/quadruplication	223
16.3.2 CRTP	224
16.3.3 Forwarding out of lambdas	225
16.3.4 Recursive lambdas	226
16.3.5 Pass this by value	227
16.3.6 SFINAE-unfriendly callables	228
16.4 Conclusion	229
17 Using the C++23 std::expected type	230
17.1 Why do we need std::expected?	230
17.1.1 Alternative 1: status code + reference parameter	230
17.1.2 Alternative 2: using exceptions	231
17.1.3 Alternative 3: using std::variant	232
17.1.4 Alternative 4: using std::optional	233
17.2 Enter std::expected	234
18 C++23: Consteval if to make compile time programming easier	238
18.1 What is if consteval?	238
18.2 How to call consteval functions?	239
18.3 Conclusion	241
19 C++23: Narrowing contextual conversions to bool	242

19.1	A quick recap	242
19.1.1	static_assert	242
19.1.2	constexpr if	243
19.2	Narrowing	243
19.3	What is the paper proposing	243
19.4	Conclusion	244
20	6 C++23 features improving string and string_view	246
20.1	std::string and std::string_view have contains	246
20.2	No more undefined behaviour due to construction from nullptr	247
20.3	Build std::string_view from ranges	247
20.4	basic_string::resize_and_overwrite()	248
20.5	Require span & basic_string_view to be TriviallyCopyable	248
20.6	string-stream with std::span-based buffer	249
20.7	Conclusion	249
21	C++23: Preprocessing directives	250
21.1	What is a preprocessing directive?	250
21.2	#elifdef and #elifendef	251
21.3	#warning	251
21.4	Conclusion	251
22	C++23: The stacktrace library	252
22.1	Is it already available? - the meta how	252
22.2	What are the key features of the stacktrace library?	252
22.3	Conclusion	256
23	C++23: flat_map, flat_set, et al.	257
23.1	Specialties of a flat_{map set}	258
23.2	One more word on speed	258
23.3	How it differs in its API	259
23.4	What if you want to use the new adaptors?	259
23.5	Conclusion	260
24	C++23: How lambdas are going to change?	261
24.1	Make () more optional for lambdas	261
24.2	Change the scope of lambda trailing-return-type	262
24.3	Attributes on lambdas	262
24.4	Conclusion	263
25	C++23: attributes	264
25.1	DR: Allow Duplicate Attributes	264
25.2	Attributes on lambdas	265
25.3	Attribute [[assume]]	265
25.4	Conclusion	266

26 A Gentle Introduction to <code>m/span</code>	267
26.1 What is <code>m/span</code> ?	267
26.2 Extents	268
26.3 Layout and access customization	269
26.4 Basics of <code>LayoutPolicy</code>	269
26.5 Views and ownership	270
27 C++23's new function syntax	272
28 Functional exception-less error handling with C++23's optional and expected	275
28.1 Why did something fail?	277
28.2 Noisy error handling	278
28.3 A theoretical aside	279
28.4 A note on overload sets	280
28.5 Try them out	280
29 The evolution of enums	282
29.1 Unscoped enumerations	282
29.2 Scoped enumerations	284
29.3 What else	285
29.4 Using-enum-declaration since C++20	286
29.5 C++23 brings <code>std::is_scoped_enum</code>	287
29.6 C++23 introduces <code>std::to_underlying</code>	287
29.7 Conclusion	288
IV Metaprogramming	289
30 C++ 反射 第一章 通识	290
30.1 C++ 中的产生式元编程	290
30.2 反射的相关概念	291
30.3 C++ 中的反射	293
30.4 百花齐放	294
30.5 总结	295
31 C++ 反射 第二章 探索	296
31.1 动态反射 & 静态反射	296
31.2 Boost Describe Library	296
31.3 RTTR Library	298
31.4 Cista Library	300
31.5 iguana Library	301
31.6 Schema Generation	303
31.7 总结	307
32 C++ 反射 第三章 原生	308

32.1	基本 Introspection	308
32.2	自定义 Attributes	312
32.3	Typed enums	314
32.4	动态命名	317
32.5	动态生成类	317
32.6	总结	319
33	C++ 反射 第四章 标准	320
33.1	C++ 静态反射与元编程的关系	320
33.2	实践环境的选择	321
33.3	The ^ operator and Splicing	321
33.4	标准元编程库	323
33.5	源码注入	327
33.6	自动生成 getters 和 setters	330
33.7	自动生成 SQL 语句	332
33.8	总结	333
V	Concurrency	334
34	STRUCTURED CONCURRENCY IN C++	335
34.1	Structured programming	336
34.1.1	Use of abstractions	336
34.1.2	Recursive decomposition of programs	337
34.1.3	Local reasoning and nested scopes	337
34.1.4	Single entry, single exit point for code blocks	338
34.1.5	Soundness and completeness	338
34.2	Concurrency with threads and synchronisation primitives	339
34.3	Concurrency with raw tasks	339
34.4	Concurrency with senders/receivers	340
34.4.1	Use of abstractions	341
34.4.2	Recursive decomposition of programs	342
34.4.3	Local reasoning and nested scopes	342
34.4.4	Single entry, single exit point	342
34.4.5	Soundness and completeness	343
34.4.6	Bonus: coroutines	343
34.5	Discussion	345
34.6	conclusions	345
35	Comparing Rust's and C++'s Concurrency Library	347
35.1	atomic_ref	347
35.2	Generic atomic type	348
35.3	Compare-exchange with padding	348
35.4	Compare-exchange memory ordering	348

35.5	constexpr Mutex constructor	349
35.6	Latches and barriers	349
35.7	Semaphore	349
35.8	Atomic wait and notify	350
35.9	jthread and stop_token	350
35.10	Atomic floats	351
35.11	Atomic per byte memcpy	351
35.12	Atomic shared_ptr	352
35.13	synchronized_value	352
35.14	Conclusion	352
36	C++ 20 concurrency	353
36.1	Part 1: synchronized output stream	353
36.1.1	The Problem	353
36.1.2	The solution	354
36.1.3	Underlying mechanism	354
36.1.4	Synchronized output stream for files	355
36.1.5	Conclusion	356
36.2	Part 2: jthreads	356
36.2.1	std::jthread	356
36.2.2	Swapping two std::jthread	358
36.2.3	Conclusion	359
36.3	Part 3: request_stop and stop_token for std::jthread	359
36.3.1	Introduction: Two ways to cooperatively stop the thread	359
36.3.2	Using std::jthread internal stop-state	360
36.3.3	Summary	364
37	co_resource<T>: An RAII coroutine	366
37.1	Language Envy	368
37.2	Implementing co_resource	371
37.2.1	Getting Started with Coroutines	371
37.2.2	The Return Type and the Promise Type	372
37.2.3	The Promise Object and the Coroutine State	372
37.2.4	Some Boilerplate	373
37.2.5	Yielding a Value	374
37.2.6	The Return Object	374
37.3	What Else?	377
38	C++ Coroutines: Understanding the Compiler Transform	378
38.1	Introduction	378
38.2	Setting the Scene	379
38.3	Defining the task type	379
38.4	Step 1: Determining the promise type	380
38.5	Step 2: Creating the coroutine state	381

38.6 Step 3: Call <code>get_return_object()</code>	384
38.7 Step 4: The initial-suspend point	384
38.8 Step 5: Recording the suspend-point	387
38.9 Step 6: Implementing <code>coroutine_handle::resume()</code> and <code>coroutine_handle::destroy()</code>	388
38.10 Step 7: Implementing <code>coroutine_handle<Promise>::promise()</code> and <code>from_promise()</code>	390
38.11 Step 8: The beginnings of the coroutine body	393
38.12 Step 9: Lowering the <code>co_await</code> expression	395
38.13 Step 10: Implementing <code>unhandled_exception()</code>	398
38.14 Step 11: Implementing <code>co_return</code>	404
38.15 Step 12: Implementing <code>final_suspend()</code>	404
38.16 Step 13: Implementing symmetric-transfer and the noop-coroutine	406
38.17 One last thing	410
38.18 Tying it all together	411
39 Back Porting C++20 Coroutines to C++14 – A Tutorial	417
39.1 Coroutine Usage	418
39.1.1 <code>co_await</code>	418
39.1.2 <code>co_yield</code>	418
39.1.3 <code>co_return</code>	419
39.2 The Coroutine Transformation	420
39.3 Issues and Solutions	422
39.3.1 <code>goto suspent_point</code>	422
39.3.2 Local variables	423
39.3.3 <code>std::coroutine_handle<promise_type></code>	423
39.3.4 Allocations	424
39.4 Macros for Readability	424
39.4.1 <code>CO_BEGIN(return_type, ...)</code>	425
39.4.2 <code>CO_AWAIT(RETURN_VAL, EXPR)</code>	425
39.4.3 <code>CO_END()</code>	426
40 Debugging C++ Coroutines	428
40.1 Introduction	428
40.2 Terminology	428
40.2.1 <code>coroutine type</code>	428
40.2.2 <code>coroutine</code>	428
40.2.3 <code>coroutine frame</code>	429
40.3 The structure of coroutine frames	429
40.4 Print <code>promise_type</code>	429
40.5 Print coroutine frames	430
40.5.1 Examples to print coroutine frames	430
40.6 Get the suspended points	433
40.7 Get the asynchronous stack	434

40.7.1	Examples to print asynchronous stack	434
40.8	Get the living coroutines	443
41	C++20 Coroutines and io_uring	444
41.1	C++20 Coroutines and io_uring - Part 1/3	444
41.1.1	Async I/O Coroutines Permalink	444
41.1.2	Goals	445
41.1.3	Ground Work	445
41.1.4	First Attempt: A Trivial Approach	446
41.1.5	Next Attempt: Thread Pool	447
41.1.6	Enter io_uring	448
41.1.7	Parsing OBJS with liburing	448
41.1.8	Closing Thoughts	451
41.2	C++20 Coroutines and io_uring - Part 2/3	452
41.2.1	Basic Idea	452
41.2.2	Suspending Execution	452
41.2.3	Resuming Execution	454
41.2.4	Coroutine Type: Task	455
41.2.5	Putting it all together	457
41.2.6	Closing Thoughts	458
41.3	C++20 Coroutines and io_uring - Part 3/3	458
41.3.1	ThreadPool	458
41.3.2	Multi-Threaded Implementation	459
41.3.3	Closing Words	460
42	Instruction-level parallelism in practice: speeding up memory-bound programs with low ILP	461
42.1	A Quick Introduction to Instruction-Level Parallelism	461
42.2	Analyzing available instruction-level parallelism	462
42.2.1	Code Example with High ILP	462
42.2.2	Code Example with Low ILP, but no Dependency Chains in Memory Loading	463
42.2.3	Code Example with Low ILP and Dependency Chains on Memory Loading	464
42.3	What Are the Codes with Low ILP?	464
42.4	Techniques Used to Increase Available Instruction-Level Parallelism	465
42.4.1	Interleaving Additional Work	466
42.4.2	Breaking Pointer Chains	467
42.4.3	Shortening the Dependency Chain	468
42.5	Experiments	469
42.5.1	Interleaving Lookups in a Binary Tree	469
42.5.2	Breaking Dependencies with Array-Based Binary Tree	470
42.5.3	Breaking Dependencies in a Linked List	471
42.6	Conclusion	473

VI Performance	474
43 5 Ways to Initialize a String Member	475
43.1 call by-const-reference	475
43.2 call by-value	476
43.3 Two-overloads	476
43.4 C++17 string_view	477
43.5 Fowarding references	477
43.6 Long String 成员开销对比	478
43.7 SSO 短字符串优化	479
43.8 无拷贝，无移动	480
43.9 优化限制：Aliasing situations	480
43.10 总结	481
44 Using final in C++ to improve performance	482
44.1 Interfaces and subtyping	482
44.1.1 Simple Interface Example	482
44.2 The cost of Dynamic Polymorphic behaviour	486
44.2.1 Using polymorphic functions	488
44.3 Introducing final	489
44.4 Devirtualization	491
44.4.1 Templates and final	491
44.4.2 Revising the Interface	492
44.5 Summary	494
45 Technique: Compile Time Code Generation and Optimization	495
45.1 Brainfuck	495
45.2 Step 1: A traditional Brainfuck VM	495
45.3 Step 2: Tail recursion	499
45.4 Step 3: Making it a template	500
45.5 Conclusion	502
46 Improving the State of Debug Performance in C++	503
46.1 Overview	503
46.2 Motivation	503
46.3 Show me some code!	504
46.4 How we did it	516
46.5 Looking ahead...	516
46.5.1 Closing	516
47 Simplify Code with if constexpr and Concepts in C++17/C++20	517
47.1 Intro	517
47.2 Why compile-time if?	518
47.3 std::enable_if	519

47.3.1	Use Case 1 - Comparing Numbers	520
47.3.2	Use case 2 - computing the average	523
47.3.3	Use case 3 - a factory with variable arguments	524
47.3.4	Use case 4 - real-life projects	528
47.4	Wrap up	529
47.4.1	Going back...	530
47.4.2	Even more	530
48	The memory subsystem from the viewpoint of software: how memory subsystem affects software performance	531
48.1	Part 1	531
48.1.1	A short introduction to the memory hierarchy	531
48.1.2	The experiments	533
48.1.3	Final Words	542
48.2	Part 2	543
48.2.1	The Experiments	543
48.2.2	Hiding Memory Latency	546
48.2.3	TLB cache and large pages	549
48.2.4	Cache Conflicts	549
48.2.5	Vectorization	551
48.2.6	Branch Prediction	554
48.2.7	Final Words	556
49	the sad state of debug performance in c++	557
49.1	moving an int is slow	557
49.2	it gets worse...much worse	558
49.3	what are the consequences?	559
49.4	using optimizations in debug mode	560
49.5	what can be done?	560
49.6	faq	561
49.7	conclusion	562
50	Using Requires Expression in C++20 as a Standalone Feature	563
50.1	static_assert	563
50.2	constexpr if	564
50.3	Requires Clause	566
50.4	'requires requires' or anonymous concepts	569
51	What is faster: vec.emplace_back(x) or vec[x] ?	571
51.1	The Experiment	571
51.2	First Results	572
51.3	Question 1: Why is vector initialization so slow for operator[] ?	572
51.4	Question 2: Why is the hot loop in emplace_back() slower? Pagefaults or something else?	573
51.5	Question 3: Why is only the operator[] hot loop vectorized?	574

51.6	Question 4: What is the effect of pagefaults on runtime for the two hot loops?	574
51.7	Question 5: If the operator[] hot loop is not vectorized, which version is faster then?	575
51.8	Discussion	575
52	P2723R0 Zero-initialize objects of automatic storage duration	576
52.1	Summary	576
52.2	Opting out	578
52.3	Security details	578
52.4	Alternatives	580
52.5	Performance	582
52.6	Caveats	583
52.7	Wording	584
52.7.1	References	584
52.7.2	Informative References	584
53	wordexpr: compile-time wordle in C++20	587
53.1	high-level overview	587
53.2	error is the new printf	587
53.3	compile-time random number generation	589
53.4	retaining state and making progress	589
53.5	conclusion	590
54	const vs constexpr vs consteval vs constit in C++20	591
54.1	const vs constexpr	591
54.2	constexpr vs consteval	593
54.3	constexpr vs constit	594
54.4	Mixing	595
54.5	Summary	595
VII	Algorithms	597
55	C++20 Ranges Algorithms	598
55.1	7 Non-modifying Operations	598
55.1.1	Before we start	598
55.1.2	all_of, any_of, none_of	599
55.1.3	for_each	600
55.1.4	count_if	602
55.1.5	find_if	602
55.1.6	find_first_of	604
55.1.7	mismatch	606
55.1.8	search	607
55.1.9	Summary	610
55.2	11 Modifying Operations	610

55.2.1	copy_if	610
55.2.2	fill	611
55.2.3	generate	612
55.2.4	transform	613
55.2.5	remove	615
55.2.6	replace	616
55.2.7	reverse	618
55.2.8	rotate	619
55.2.9	shuffle	621
55.2.10	sample	621
55.2.11	unique	623
55.2.12	Summary	624
55.3	sorting, sets, other and C++23 updates	624
55.3.1	Partitioning & Sorting	624
55.3.2	Binary Search operations	627
55.3.3	Set operations	628
55.3.4	Other	630
55.3.5	Numeric	633
55.3.6	Soon in C++23	633
55.3.7	Summary	633
56	New Fold Algorithms	634
56.1	Background: Rangified Algorithms	634
56.2	accumulate and reduce	635
56.3	std::ranges::fold_*	636
56.3.1	fold_left	636
56.3.2	fold_right	636
56.3.3	Aside on initial element	637
56.3.4	fold_left_first and fold_right_last	637
56.3.5	fold_left_with_iter and fold_left_first_with_iter	638
56.4	What about reduce?	638
56.5	What about projections?	639
56.6	Feedback	639
56.7	Acknowledgements	639
57	Faster integer formatting - James Anhalt (jeaiii)' s algorithm	640
57.1	Disclaimer	640
57.2	Naïve implementations	641
57.3	The core idea of James Anhalt' s algorithm	643
57.4	How to compute y ?	644
57.5	Consideration of variable length	646
57.6	Better choices for y	651
57.7	Benchmark	656

57.8	Back to fixed-length case	656
57.9	Concluding remarks	657
VIII	Techniques	658
58	Conditional Members	659
58.1	Conditional Member Functions	659
58.2	Conditional Member Types	662
58.3	An alternate approach	666
59	Inside boost::unordered_flat_map	669
59.1	Introduction	669
59.2	The case for open addressing	669
59.2.1	SIMD-accelerated lookup	671
59.3	Rehashing	672
59.4	Hash post-mixing	673
59.5	Statistical properties of boost::unordered_flat_map	673
59.6	Benchmarks	675
59.6.1	Running-n plots	675
59.6.2	Aggregate performance	676
59.7	Deviations from the standard	678
59.8	Conclusions and next steps	678
60	Static B-Trees	679
60.1	B-Tree Layout	680
60.2	Implicit B-Tree	680
60.3	Construction	681
60.4	Searches	681
60.5	Optimization	683
60.6	B+ Tree Layout	687
60.7	Implicit B+ Tree	688
60.8	Construction	689
60.9	Searching	690
60.10	Comparison with std::lower_bound	690
60.11	Modifications and Further Optimizations	693
60.12	As a Dynamic Tree	695
60.13	Acknowledgements	696
61	Technique: Recursive variants and boxes	697
61.1	The problem	697
61.2	Heap allocating nested expressions	698
61.3	Adding value semantics	700
61.4	Aside: Moving boxes	702

61.5	Conclusion	703
62	Working with Strings in Embedded C++	704
62.1	Embedded systems and strings	704
62.1.1	Character literals	705
62.2	C Strings	705
62.2.1	Null-Terminated Byte Strings (NTBS)	705
62.2.2	C-Strings and string literals	706
62.2.3	Finding strings	707
62.2.4	C String Standard Library	708
62.2.5	Safety and Security Issues	708
62.3	C++ Strings	708
62.3.1	C++11 Raw String Literals	708
62.3.2	C++ Strings Library	709
62.3.3	Numeric to std::to_string	710
62.3.4	std::string and NTBS	711
62.3.5	typedefs for string types	712
62.4	String memory management	712
62.4.1	Strings are, by default, heap-allocated	712
62.4.2	Copying Strings	713
62.4.3	Deep copy of std::string	714
62.4.4	Moving std::string	714
62.4.5	Short String Optimisation (SSO)	715
62.4.6	SSO -GCC	716
62.4.7	String APIs	717
62.5	C++17 std::string_view	718
62.5.1	Constant pointer + size	719
62.5.2	std::string_view as a NTBS replacement	720
62.5.3	Common APIs with string-like syntax	720
62.5.4	auto type deduction	721
62.5.5	auto vs decltype	722
62.6	std::string_view Caveats	723
62.6.1	string lifetime management	723
62.6.2	non null-terminated strings	724
62.7	Polymorphic Memory Resource (PMR) Strings	725
62.8	Summary	726
63	Determining if a template specialization exists	728
63.1	Testing for a specific function template specialization	728
63.2	Testing for a specific class template specialization	729
63.3	A generic test for class templates	730
63.4	Problem: Specializations that sometimes exist and sometimes don't	732
63.5	A generic test for function templates	733

63.6	What do I mean by “a specialization exists”	734
63.7	A note on MSVC and std::hash	735
64	How to Store an lvalue or an rvalue in the Same Object	737
64.1	Keeping track of a value	737
64.1.1	Storing a reference	737
64.1.2	Storing a value	739
64.2	Storing a variant	740
64.3	A generic storage class	740
64.3.1	Defining const access	741
64.3.2	Defining non-const access	742
64.4	Creating the storage	743
64.5	The evolution of the standard library	743
65	Automatic Serialization in C++ for Game Engines	745
65.1	Caveats	745
65.1.1	References and Pointers can’t be serialised	745
65.1.2	Objects need to be default constructible and copyable	747
65.1.3	You have to add some special code to your classes	747
65.2	Why use Preprocessor Macros for C++ Serialization?	748
65.3	Building Blocks	749
65.4	Looping Preprocessor Macro	752
65.5	Operator » and operator « Macro Replacements	754
65.6	Next Steps	758
66	Technique: Take a <code>constexpr</code> string from Compile Time to Run Time	759
IX	Tricks	765
67	如何优雅地打印类型名称?	766
67.1	概述	766
67.2	编译期打印类型名称	766
67.3	Demangled Name	767
67.4	编译器扩展特性	770
67.5	Circle	772
67.6	Static Reflection	773
67.7	总结	773
68	Avoiding direct syscall instructions by using trampolines	774
68.1	Detecting direct syscalls made through SysWhispers	775
68.2	Retrieve syscalls with HellsGate	775
68.3	Make it bounce!	776
69	Avoiding Temporaries with Expression Templates	778

69.1	A first naive Approach	778
69.2	Expression templates	781
69.3	Under the hood	785
70	Casting a negative float to an unsigned int	786
70.1	Introduction	786
70.2	How the Cast Became Undefined	787
70.3	Why the Cast Fails	787
70.4	How to Fix the Problem	788
70.5	Earlier Feedback Needed	788
71	Compiler Tricks to Avoid ABI-Induced Crashes	790
71.1	Attempt #1: NOINLINE	791
71.2	Attempt #2: optimize off	791
71.3	Attempt #3: it's clobbering time	791
71.4	Clobber blocks	792
71.5	Fixed, but should we?	793
72	Five tricky topics for data members in C++20	794
72.1	1. Changing status of aggregates	794
72.2	2. No parens for direct initialization and NSDMI	796
72.3	3. No deduction for NSDMI	797
72.4	4. List initialization. Is it uniform?	798
72.5	5. std::initializer_list is greedy	799
72.6	Summary	800
73	On finding the average of two unsigned integers without overflow	801
X	Miscellaneous	811
74	New integer types I'd like to see	812
74.1	Symmetric signed integers	812
74.2	Unsigned integers with one bit missing	813
74.3	Distinct bit vectors vs integer type	814
74.4	Conclusion	815
75	proxy: Runtime Polymorphism Made Easier Than Ever	816
75.1	Overview	816
75.2	Configure your project	818
75.3	What makes the “proxy” so charming	818
75.4	Highlight 1: Being non-intrusive	819
75.5	Highlight 2: Evolutionary lifetime management	820
75.6	Highlight 3: High-quality code generation	823
75.7	Highlight 4: Composition of abstractions	823

75.8	Highlight 5: Syntax for CPOs and modules	824
75.9	Highlight 6: Static reflection	824
75.10	Highlight 7: Performance tuning	825
75.11	Highlight 8: Diagnostics	826
75.12	Conclusion	827
76	C23 is Finished: Here is What is on the Menu	828
76.1	N3006 + N3018 - constexpr for Object Definitions	829
76.2	N3038 - Introduce Storage Classes for Compound Literals	831
76.3	N3017 - #embed	834
76.4	N3033 - Comma Omission and Deletion (<code>__VA_OPT__</code> in C and Preprocessor Wording Improvements)	834
76.5	N2975 - Relax requirements for variadic parameter lists	835
76.6	N3029 - Improved Normal Enumerations	836
76.7	N3030 - Enhanced Enumerations	837
76.8	N3020 - Qualifier-preserving Standard Functions	840
76.9	N3042 - Introduce the <code>nullptr</code> constant	841
76.10	N3022 - Modern Bit Utilities	842
76.11	N3006 + N3007 - Type Inference for object definitions	843
76.12	N2897 - <code>memset_explicit</code>	844
76.13	N2888 - Exact-width Integer Types May Exceed <code>(u)intmax_t</code>	845
76.14	And That's All I'm Writing About For Now	845
77	Cpp2 and cppfront: Year-end mini-update	847
77.1	10 design notes	847
77.2	117 issues (3 open), 74 pull requests (9 open), 6 related projects, and new collaborators	848
77.3	Compiler/language improvements	849
77.4	What's next	850
78	Copy-Paste Developments	851
78.1	Copy-paste developments	851
78.2	The need to understand	852
78.3	Dead code	852
78.4	You'll need to understand it anyway	852
78.5	Extending your known scope	853
78.6	Intellectual interest	853
78.7	Not copy-pasting	853
78.8	Don't let your fingers muscles do all the work	853
79	Carbon's most exciting feature is its calling convention	854
79.1	Carbon's parameter passing	854
79.2	Advantage #1: Performance	855
79.3	Advantage #2: Optimal calling convention in generic code	856
79.4	Advantage #3: Copies that aren't copies	856

79.5	Advantage #4: Parameters without address	856
79.6	More precise escape analysis	857
79.7	Explicit address syntax	858
79.8	Future language extensions	859
79.9	Conclusion	860
80	Ways to Refactor Toggle/Boolean Parameters in C++	862
80.1	Intro	862
80.2	ideas	863
80.2.1	Small enums	863
80.2.2	Bit flags	864
80.2.3	Param structure	865
80.2.4	How about C++20	865
80.2.5	Elimination	866
80.2.6	Stronger types	867
80.2.7	C++ Guidelines	867
80.2.8	Tools	868
80.3	A concrete example	868
80.4	Summary	870
81	malloc() and free() are a bad API	871
81.1	The C allocation functions	871
81.2	Problem #1: Alignment	872
81.3	Problem #2: Metadata storage	872
81.4	Problem #3: Wasting space	874
81.5	Problem #4: realloc()	874
81.6	A better interface	875
81.7	C++ Solutions	875
81.8	C++17: Aligned allocation	876
81.9	C++17: Sized deallocation	876
81.10	C++23: Size feedback in std::allocator	876
81.11	Conclusion	877
82	Use compiler flags for stack protection in GCC and Clang	878
82.1	Stack canary	878
82.2	SafeStack and shadow stack	878
82.3	Fortified source	879
82.4	Control flow integrity	879
82.5	Stack allocation control	880
82.6	Stack usage and statistics	880
82.7	Summary	881
83	The Wonderfully Terrible World of C and C++ Encoding APIs (with Some Rust)	882
83.1	Enumerating the Needs	883

83.1.1	Handles Legacy Encodings	885
83.1.2	Handles UTF Encodings	886
83.1.3	Safe Conversion API	886
83.1.4	Assumed Valid Conversion API	886
83.1.5	Unbounded Conversion API	886
83.1.6	Counting API	887
83.1.7	Validation API	887
83.1.8	Extensible to (Runtime) User Encodings	887
83.1.9	Bulk Conversions	887
83.1.10	Single Conversions	887
83.1.11	Custom Error Handling	888
83.1.12	Updates Input Range / Updates Output Range	888
83.1.13	Nevertheless	889
83.2	ICU	889
83.3	simdutf	890
83.3.1	One, The Other, But Definitely Not Both.	892
83.3.2	A Brief & Ranting Segue: Output Ranges and “Modern” C++	893
83.3.3	Rant Over	896
83.4	utf8cpp	896
83.5	encoding_rs/encoding_c	897
83.6	libiconv	900
83.7	boost.text	902
83.8	Standard C and C++	903
83.9	Windows API	905
83.10	ztd.text	907
83.11	So...What Happens Now?	907
83.12	... Part 2	910

This page is intentionally left blank.

序

技术提高关键有三，一是输入，二是理解，三是输出。后二者皆受输入质量影响，低者产生浅理解弱输出，高者产生深理解强输出。是以书籍择取于学习者而言，至关重要。

去年整理数百篇文章，主题包罗万象，靡不讨论，质量却是参差不齐，便有筛选之心。本不堪繁琐，然积累既久，优文众多，不乏可反复阅读者，按题选类，裒然成帙，于己于人俱是有益。故召集数群友实现之，去芜存菁，细复排版，历三月有余，遂成此书。

本书含十大主题，分为 Basics、Modern C++、C++23、Metaprogramming、Concurrency、Performance、Algorithms、Techniques、Tricks 和 Miscellaneous。广度少有能及，又因作者众多，深度亦有保证，加之内容甚新，录有 C++26/29 才有可能加入的特性，可谓是当前最新之书矣。

既致之以广大，又尽之以精微，合之以主题，以成本书内容之深广也。而每章独立，各有所论，故本书针对高级开发者。饶是如此，新手亦可依此进阶，小大靡遗，纤悉洞了，查漏补缺，得窥其理。

本书 Modern C++ 部分由 404 整理，Performance 由何荣华整理，Techniques 与 Tricks 由水滴会飞整理，Miscellaneous 由邹启翔整理，其余部分皆由我负责。谨致谢意！

因时间仓促，间隔颇长，且限于所识，错误之处在所难免，敬请高明者匡正。

2023 年 5 月 5 日
里缪

写作结构

写作结构，分为两种，一种是自上而下，一种是自下而上。

先说其一。此种先是给出高级的概念，再分而述之，结构清晰，具有较好的可预知性。偏科普，告诉读者何为有效，何为无效，读者无需再费力去整理脉络。

例如，一篇讲解某个算法的文章。此种方式，第一段会先说明针对某个问题，存在几种解决方式，名称一、名称二、名称三……第二段介绍第一种解决方式，概念一、概念二、概念三……第三段说明第一种解决方式的优缺点，理由一、理由二、理由三……第四段介绍第二种方式，对比方式一，优点一、优点二、优点三……第五段，补充说明，归纳总结。

这种是知识掌握者以自己构建好的知识体系，向读者抛出知识点。故省略了诸多作者在学习时遇到的问题，掐掉了逻辑链的中间环节，只向读者展示一头一尾。因此于知识薄弱的读者而言，晦涩难懂，理解费劲，在所难免。结果引出，分述方法，点明优劣，补充说明，重述总结。于知识储备较强的人而言，这种文章结构清晰，文字精炼，利于结构化。

再说其二。此种先提出具体例子，复慢引出抽象概念，循序渐进，逻辑紧密。易让人对事物产生直观认识，再自己去整理脉络，可加深记忆。偏研究，宜突出概念间的联系。

相同例子。此种方式，第一段会先给出某个现实例子，引出实际存在的问题；第二段牵出能够解决该问题的思路，然后点出方式中的种种东西，其实就叫术语一、术语二、术语三……第三段指出这种思路的不足，原因一、原因二、原因三……第四段再给出进一步的解决思路，此时再告诉你这种解决思路其实就谓之什么方式；第五段，补充说明，归纳总结。

学习就是自下而上，不断归纳的过程，此种方式重现了这个过程，保留了逻辑链的所有环节，正因如此，读来才觉通俗易懂。教学由于受众广，采用这种方式更有利于知识传播。提出问题，引出分析，牵出结果，指出优劣，复加改善，对比结果，总结归纳。于知识储备较弱的人而言，这种文章更易于理解和吸收，但文字琐碎，不利于结构化。

理论型和技术型文章基本都是采用这两种写作结构，比如数学教材一般采取自上而下的方式，学术论文一般也采用自上而下的方式，国外一些作者写的通俗博客一般则采用自下而上的方式。本书每章内容亦不例外，其实单从目录来看，就能分析出每章的写作结构，从而能够更加高效地阅读。

顺便一提，本部分本身就是自上而下的结构。

2023年5月6日

里缪

难度等级

难度等级是区分文章难度与质量的一种方式，分为五级：

- 一颗星：Easy（简单）
- 二颗星：Normal（普通）
- 三颗星：Hard（困难）
- 四颗星：Challenging（挑战）
- 五颗星：Fiendish（极难）

如何评判一篇文章的难度等级呢？主要在于联系二字。

一星之文，常只含单个概念，内容简单，篇幅短小，读来轻而易举。

二星之文，常含三两概念，皆属同一主题，难度不大，理解轻松。一星二星之文几乎不含作者观点，仅是简单罗列概念，缺乏区分度和唯一性。

三星之文，能够覆盖某个主题的绝大多数概念，讲清概念用法、发展历程，但浅尝辄止。多数文章都属此类，作者对内容进行初步加工、归纳总结，再以某种逻辑链将零散的东西串起来，以清晰的形式呈现给读者。

四星之文，能够思考某个主题下概念间的联系，剖析入微，鞭辟入里，理清平时不易注意的细枝末节。该层级的内容经过作者的深度加工，几乎完全覆盖了某个主题下概念间的来龙去脉。

五星之文，主题宽广，概念繁多，错综复杂，理解颇难。此类文章聚焦于庞大主题的某个局部网络，把不同主题的概念放到一起，依照其内在相似点和共性，总结出一个新的、更高层次的结论。内容博大精深，讲解面面俱到，此文寥寥。

一星文只涉及某个点，二星文涉及几个点，三星文由点组成线，四星文对这条线进行横向和纵向的深度挖掘，五星文由线构成网，不仅主题宽广，概念繁多，而且内容够深，联系紧密。一星二星之文，内容未经作者加工，只是一些原封不动照搬的信息而已，其实更应称之为笔记；三星开始，作者才把自己的思想赋予文章，内容不再穿凿，结构清晰，逻辑完整；四星则在三星基础上更深一步，除了内容间的连贯性，概念本身也讨论的更加深刻；五星是跨主题间的概念讨论，能够跨主题的概念本身就极其复杂，再要将其串起，实非易事，故五星之文写来费时费力。写文一周，积累数月，此之谓也。是以一二星文最多，三四星次之，五星最少。

除 C++23，本书基本不会包含一二星文，23 的文章在于一个新，难度暂可忽略。我们整理的文章主要集中于三星到五星，其中也是以三四星居多，五星文章很少。文章评星，易带主观，故仅作参考，实际难度可能相差一星。

2023 年 5 月 9 日
里缪

错误反馈

发现任何格式或其他错误，请到<https://github.com/lkimuk/the-book-of-modern-cpp/issues>反馈。

This page is intentionally left blank.

Part I

Basics

一个语言中最容易被忽视的就是一些最基本的概念，属于是新手学皮毛，老手看不上。然而对语言的深入理解，往往需要在某个阶段重新回归到那些最简单的概念上来。

为什么呢？知识是有网络效应的，单个知识点的价值，会随着你懂得的更多东西而提升。初学之时，你可能只懂 10 个知识点，它也许只包含 5、6 个联系，但在学习几年后，你懂得了 50 个知识点，它就可能构建出 30、40 个联系。最简单的这些概念，往往也是能够产生联系最多的概念，它可能与不同主题中的概念都有关联，那么通过它你就能够从一个更底层的逻辑链构建起许多更高层的知识点。

正因如此，Basics 的内容可深可浅。互联网上充斥着初级文章，却缺乏这种深入讨论的文章，即便是本书，也仅包含了 3 篇能归类到该主题的文章，它们讨论的概念分别是重载决议、移动语义和 ABI。

虽是第一 Part，这部分内容的难度等级绝对不低，需要一定的知识积累才能真正读懂。也就是说，这部分是给老手读的，新手暂时可跳过。

2023 年 5 月 13 日
里缪

Chapter 1

洞悉函数重载决议

• 里缪 2022-09-04



大家可以尝试问自己一个问题：

调用一个重载函数，编译器是如何找到最佳匹配函数的？

若是你不能清楚地表述这个流程，就说明对函数重载决议缺乏认识。

函数重载决议也的确是许多 C++ 开发者都听过，但却从来没有真正理解过的一个概念，基本上也没有书籍深入讲解过这一重要概念，然而对语言的深刻理解往往就是建立在对这些基本概念的理解之上。

那么理解这个有什么用呢？

对于库作者，理解重载决议必不可少。因为重载涉及函数，函数又是变化的最小单元之一，可以说重载决议贯穿了「定制点」的发展历程。只有理解重载决议，才能理解各种定制点的表现方式，比如 ADL 二段式、CPOs、Deducing this。

对于使用者，若是不理解重载决议，就不能理解定制点，也就无法真正理解各种库的设计思路，使用起来难免会束手束脚。

同时，重载决议还涉及 ADL、TAD、SFINAE、Concepts、Forwarding reference 等等概念，若不理解重载决议，对这些相关概念的理解也将难以深入。

总而言之，重载决议与 C++ 中的许多概念都有着千丝万缕的联系，且重载解析本身就是一件非常复杂的工作，这也是其难度颇高的原因。

接下来，就让我们拨开重重云雾，一探究竟。

1.1 重载决议的基本流程

函数的标识主要分为两部分，名称和参数。

当函数名称唯一时，调用过程相对简单，直接查找即可。C 语言就属此列，它的函数名称必须唯一。

当函数名称相同，但参数类型不同时，在许多语言中依旧合法，此时这些名称相同的函数就称为重载函数。

C++ 就是支持重载函数的语言之一，那么它要如何来确定函数名的唯一性？

实际上，编译器会通过一种称为 Name mangling（名称修饰）的技术来为每个重载函数生成唯一的名称。虽然重载函数的名称是相同的，但其参数不同，因此通过名称 + 参数再辅以一些规则，生成唯一的名称其实并非难事。

但这仍非实现重载函数的关键与难点所在。名称是唯一产生了，但是用户并不知道，也并不能直接通过该名称来调用函数。用户调用的还是重载函数名称本身，此时就需要一套机制来解析实际调用的函数到底是哪个，该机制就是「重载决议」，由 C++ 标准制定。

简言之，只要遇到名称相同的函数，重载决议就会出现，用于找出最佳匹配函数。

那么问题又来了，它是如何知道存在哪些名称相同的函数？

这便是在重载决议出现之前的一项工作，称为 Name Lookup（名称查找）。

这一阶段，会根据调用的函数名称，查找函数的所有声明。若函数具有唯一的名称，那么就不会触发重载决议；若查找到多个相同的函数名称，这些函数声明就会被视为一个 overload set（重载集）。

函数又分为普通函数和函数模板，在 Name Lookup 阶段都会被查找到。但是函数模板只有实例化之后才能被使用，因此如果存在函数模板，还需要对模板进行特殊的处理，这个阶段就称为 Template Handling（模板处理）。

经过上述两个阶段的处理，得到的重载集就称为 candidate functions（候选函数），重载决议的工作就是在这些 candidate functions 中，找出最适合的那个函数。

总结一下，当你调用一个重载函数时，编译器首先会进行 Name Lookup，找出所有函数声明，然后对函数模板进行 Template Handling，实例化出模板函数，产生 candidate functions，接着重载决议出现，找出最佳匹配函数。

而实际的最佳匹配函数调用，则是通过 Name mangling 产生的函数名称完成的。

1.2 Name Lookup

首先来看第一阶段，Name Lookup。该阶段仅仅进行名称查找，并不做任何额外检查。

Name Lookup 的工作主要可以分为两大部分。

第一部分为 Qualified Name Lookup（有修饰名称查找），这主要针对的是带有命名空间的函数调用，或是成员函数。

第二部分为 Unqualified Name Lookup（无修饰名称查找），这种针对的就是普通函数的调用。

下面依次进行讨论。

1.2.1 Qualified Name Lookup

带修饰的名称查找并不算复杂，这又可以主要分为两类查找。

一类是 Class Member Lookup，表示对于类成员的名称查找；另一类是 Namespace Member Lookup，表示对于命名空间下的名称查找。

其实还可以包含枚举名称，因为它也可以使用作用域解析操作符”::”进行访问，但一法通万法，不必单独细论。

以下单独讨论主要的两类。

1.2.1.1 Class Member Lookup

类成员查找，是在访问类成员时进行名称查找的规则。

成员本质上来说还是两种类型，变量与函数。换个角度来看，成员又可分为静态成员和动态成员，静态成员可以通过”::”进行访问，动态成员可以通过”.或”->”进行访问。

也就是说，当你使用如上三种方式访问某个变量或函数时，就可能会触发 Class Member Lookup。

首先来看前者，即使用”::”访问时的规则。示例如下：

```

1 // Example from ISO C++
2
3 class X {};
4 class C {
5     class X {};
6     static const int number = 50;
7     static X arr[number];
8 };
9
10 X C::arr[number]; // #1

```

可以将 #1 处的定义从”::”拆分为前后两部分。

对于前面的名称 X 和 C，将会在其定义的命名空间进行查找，此处即为全局空间，于是查找到全局作用域下的 X 和 C 类。

对于后面的名称 arr 和 number，将会在 C 类的作用域下进行查找，它们将作为类成员进行查找。

此时就是”::”前面的类型名，告诉编译器后面的名称应该通过 Class Member Lookup 进行查找。如果搜索发现前面是个命名空间，则会在相应的作用域下查找。

由于 X 是在全局作用域下查找到的，所以并不会找到内部类 X，于是该声明会产生编译错误。

接着来看后者，关于”.”和”->”的规则。看一个简单的例子：

```

1 struct S {
2     void f() {}
3 };
4
5 S s;
6 s.f();
7 S* ps = &s;
8 ps->f();

```

此处要调用 f 函数，因为使用了”.”或”->”操作符，而操作符前面又是一个类，所以 f 的查找将直接使用 Class Member Lookup，在类的作用域下进行查找。

这种调用一目了然，查找起来也比较方便，便不在此多加着墨，下面来看另一类带修饰的名称查找。

1.2.1.2 Namespace Member Lookup

命名空间成员查找，是在访问命名空间下的元素时进行名称查找的规则。

当你使用”::”访问元素的时候，就有可能会触发 Namespace Member Lookup。

比如，当把”::”单独作为前缀时，则会强制 Name Lookup 在全局空间下进行查找。如下述例子：

```

1 void f(); // #1
2
3 namespace mylib {
4
5     void f(); // #2
6

```

```

7   void h() {
8       ::f(); // calls #1
9       f(); // calls #2
10  }
11
12 } // namespace mylib

```

此时，若是没有在全局作用域下搜索到相应的函数名称，也不会调用 #2，而是产生编译错误。若是要在外部访问命名空间内部的 f(), 则必须使用 mylib::f(), 否则 Name Lookup 会找到全局作用域下的 #1。

下面再来看一个稍微复杂点的例子：

```

1 // Example from ISO C++
2
3 int x;
4 namespace Y {
5     void f(float);
6     void h(int);
7 }
8
9 namespace Z {
10    void h(double);
11 }
12
13 namespace A {
14     using namespace Y;
15     void f(int);
16     void g(int);
17     int i;
18 }
19
20 namespace B {
21     using namespace Z;
22     void f(char);
23     int i;
24 }
25
26 namespace AB {
27     using namespace A;
28     using namespace B;
29     void g();
30 }
31
32 void h() {

```

```

33     AB::g();      // #1
34     AB::f(1);    // #2
35     AB::f('c');   // #3
36     AB::x++;    // #4
37     AB::i++;    // #5
38     AB::h(16.8); // #6
39 }
```

这里一共有 6 处调用，下面分别来进行分析。

第一处调用，#1。

Name Lookup 发现 AB 是一个命名空间，于是在该空间下查找 g() 的定义，在 29 行查找成功，于是可以成功调用。

第二处调用，#2。

Name Lookup 同样先在 AB 下查找 f() 的定义，注意，查找的时候不会看参数，只看函数名称。

然而，在 AB 下未找到相关定义，可是它发现这里还有了两个 using-directives，于是接着到命名空间 A 和 B 下面查找。

之后，它分别查找到了 A::f(int) 和 B::f(char) 两个结果，此时重载决议出现，发现 A::f(int) 是更好的选择，遂进行调用。

第三处调用，#3。

它跟 #2 的 Name Lookup 流程完全相同，最终查找到了 A::f(int) 和 B::f(char)。于是重载决议出现，发现后者才是更好的选择，于是调用 B::f(char)。

第四处调用，#4。

Name Lookup 先在 AB 下查找 x 的定义，没有找到，于是再到命名空间 A 和 B 下查找，依旧没有找到。可是它发现 A 和 B 中也存在 using-directives，于是再到命名空间 Y 和 Z 下面查找。然而，还是没有找到，最终编译失败。

这里它并不会去查找全局作用域下的 x，因为 x 的访问带有修饰。

第五处调用，#5。

Name Lookup 在 AB 下查找失败，于是转到 A 和 B 下面查找，发现存在 A::i 和 B::i 两个结果。但是它们的类型也是一样，于是重载决议失败，产生 ambiguous（歧义）的错误。

最后一处调用，#6。

同样，在 AB 下查找失败，接着在 A 和 B 下进行查找，依旧失败，于是接着到 Y 和 Z 下面查找，最终找到 Y::h(int) 和 Z::h(double) 两个结果。此时重载决议出现，发现后者才是更好的选择，于是最终选择 Z::h(double)。

通过这个例子，相信大家已经具备分析 Namespace Member Lookup 名称查找流程的能力。

接着再补充几个需要注意的点。

第一点，被多次查找到的名称，但是只有一处定义时，并不会产生 ambiguous。

```

1 namespace X {
2     int a;
3 }
4
5 namespace A {
6     using namespace X;
7 }
```

```

8
9  namespace B {
10    using namespace X;
11 }
12
13 namespace AB {
14   using namespace A;
15   using namespace B;
16 }
17
18 AB::a++; // OK

```

这里，Name Lookup 最终查找了两次 X::a，但因为实际只存在一处定义，于是一切正常。

第二点，当查找到多个定义时，若其中一个定义是类或枚举，而其他定义是变量或函数，且这些定义处于同一个命名空间下，则后者会隐藏前者，即后者会被选择，否则 ambiguous。

可以通过以下例子来进行理解：

```

1 // Example from ISO C++
2
3 namespace A {
4   struct x {};
5   int x;
6   int y;
7 }
8
9 namespace B {
10  struct y {};
11 }
12
13 namespace C {
14   using namespace A;
15   using namespace B;
16   int i = C::x; // #1
17   int j = C::y; // #2
18 }

```

先看 #1，由于 C 中查找 x 失败，进而到 A 和 B 中进行查找，发现 A 中有两处定义。一处定义是类，另一处定义是变量，于是后者隐藏前者，最终选择 int x; 这处定义。

而对于 #2，最终查找到了 A::y 和 B::y 两处定义，由于定义不在同一命名空间下，所以产生 ambiguous。

到此，对 Qualified Name Lookup 的内容就基本覆盖了，下面进入 Unqualified Name Lookup。

1.2.2 Unqualified Name Lookup

无修饰的名称查找则略显复杂，却会经常出现。

总的来说，也可分为两大类。

第一类为 Usual Unqualified Lookup，即常规无修饰的名称查找，也就是普遍情况会触发的查询。

第二类为 Argument Dependant Lookup，这就是鼎鼎大名的 ADL，译为实参依赖查找。由其甚至发展出了一种定制点表示方式，称为 ADL 二段式，标准中的 std::swap, std::begin, std::end, operator<< 等等组件就是通过该法实现的。

但是本文并不会涉及定制点的讨论，因为这是我正在写的书中的某一节内容：) 内容其实非常之多之杂，本篇文章其实就是为该节扫除阅读障碍而特意写的，侧重点并不同。我额外写过一篇介绍定制点的文章【使用 Concepts 表示变化「定制点」】，各位可作开胃菜。

以下两节，分别讲解这两类名称查找。

1.2.2.1 Usual Unqualified Lookup

普通的函数调用都会触发 Usual Unqualified Lookup，先看一个简单的例子：

```

1 void f(char);
2
3 void f(double);
4
5 namespace mylib {
6     void f(int);
7
8     void h() {
9         f(3);    // #1
10        f(.0);   // #2
11    }
12 }
```

对于 #1 和 #2，Name Lookup 会如何查找？最终会调用哪个重载函数？

实际上只会查找到 f(int)，#1 直接调用，#2 经过了隐式转换后调用。

为什么呢？记住一个准则，根据作用域查找顺序，当 Name Lookup 在某个作用域找到声明之后，便会停止查找。关于作用域的查找顺序，后面会介绍。

因此，当查找到 f(int)，它就不会再去全局查找其他声明。

注意：即使当前查找到的名称实际无法成功调用，也并不改变该准则。看如下例子：

```

1 void f(int);
2
3 namespace mylib {
4     void f(const char*);
5
6     void h() {
7         f(3);    // #1 Error
8     }
9 }
```

此时，依旧只会查找到 f(const char*)，即使 f(int) 才是正确的选择。由于没有相应的隐式转换，该代码最终编译失败。

那么具体的作用域查找顺序是怎样的？请看下述例子：

```

1 // Example from ISO C++
2
3 namespace M {
4
5     class B { // S3
6 };
7 }
8
9 // S5
10 namespace N {
11     // S4
12     class Y : public M::B {
13         // S2
14         class X {
15             // S1
16             int a[i]; // #1
17         };
18     };
19 }
```

#1 处使用了变量 i，因此 Name Lookup 需要进行查找，那么查找顺序将从 S1-S5。所以，只要在 S1-S5 的任何一处声明该变量，就可以被 Name Lookup 成功找到。

接着来看另一个查找规则，如果一个命名空间下的变量是在外部重新定义的，那么该定义中涉及的其他名称也会在对应的命名空间下查找。

简单的例子：

```

1 // Example from ISO C++
2
3 namespace N {
4     int i = 4;
5     extern int j;
6 }
7
8 int i = 2;
9 int N::j = i; // j = 4
```

由于 N::j 在外部重新定义，因此变量 i 也会在命名空间 N 下进行查找，于是 j 的值为 4。如果在 N 下没有查找到，才会查找到全局的定义，此时 j 的值为 2。

而对于友元函数，查找规则又不相同，看如下例子：

```

1 // Example from ISO C++
2
3 struct A {
4     typedef int AT;
```

```

5     void f1(AT);
6     void f2(float);
7     template <class T> void f3();
8 };
9
10 struct B {
11     typedef char AT;
12     typedef float BT;
13     friend void A::f1(AT);    // #1
14     friend void A::f2(BT);   // #2
15     friend void A::f3<AT>(); // #3
16 };

```

此处, #1 的 AT 查找到的是 A::AT, #2 的 BT 查找到的是 B::BT, 而 #3 的 AT 查找到的是 B::AT。

这是因为, 当查找的名称并非模板参数时, 首先会在友元函数的原有作用域进行查找, 若没查找到, 则再在当前作用域进行查找。对于模板参数, 则直接在当前作用域进行查找。

1.2.2.2 Argument Dependant Lookup

终于到了著名的 ADL, 这是另一种无修饰名称查找方式。

什么是 ADL? 其实概念很简单, 看如下示例。

```

1 namespace mylib {
2     struct S {};
3     void f(S);
4 }
5
6 int main() {
7     mylib::S s;
8     f(s); // #1, OK
9 }

```

按照 Usual Unqualified Lookup 是无法查找到 #1 处调用的声明的, 此时编译器就要宣布放弃吗? 并不会, 而是再根据调用参数的作用域来进行查找。此处, 变量 s 的类型为 mylib::S, 于是将在命名空间 mylib 下继续查找, 最终成功找到声明。

由于这种方式是根据调用所依赖的参数进行名称查找的, 因此称为实参依赖查找。

那么有没有办法阻止 ADL 呢? 其实很简单。

```

1 namespace mylib {
2     struct S {};
3     void f(S) {
4         std::cout << "f found by ADL\n";
5     }
6 }
7
8 void f(mylib::S) {

```

```

9     std::cout << "global f found by Usual Unqualified Lookup\n";
10 }
11
12 int main() {
13     mylib::S s;
14     (f)(s); // OK, calls global f
15 }
```

这里存在两个定义，本应产生歧义，但当你给调用名称加个括号，就可以阻止 ADL，从而消除歧义。

实际上，ADL 最初提出来是为了简化重载调用的，可以看如下例子。

```

1 int main() {
2     // std::operator<<(std::ostream&, const char*)
3     // found by ADL.
4     std::cout << "dummy string\n";
5
6     // same as above
7     operator<<(std::cout, "dummy string\n");
8 }
```

如果没有 ADL，那么 Unqualified Name Lookup 是无法找到你所定义的重载操作符的，此时你只能写出完整命名空间，通过 Qualified Name Lookup 来查找到相关定义。

但这样代码写起来就会非常麻烦，因此，Unqualified Name Lookup 新增加了这种 ADL 查找方式。

在编写一个数学库的时候，其中涉及大量的操作符重载，此时 ADL 就尤为重要，否则像是”+”，”==” 这些操作符的调用都会非常麻烦。

后来 ADL 就被广泛运用，普通函数也支持此种查找方式，由此还诞生了一些奇技淫巧。

不过，在说此之前，让我们先熟悉一下常用的 ADL 规则，主要介绍四点。

第一点，当实参类型为函数时，ADL 会根据该函数的参数及返回值所属作用域进行查找。

例子如下：

```

1 namespace B {
2     struct R {};
3     void g(...) {
4         std::cout << "g found by ADL\n";
5     }
6 }
7
8 namespace A {
9     struct S {};
10    typedef B::R (*pf)(S);
11
12    void f(pf) {
13        std::cout << "f found by ADL\n";
```

```

14     }
15 }
16
17 B::R bar(A::S) {
18     return {};
19 }
20
21 int main() {
22     A::pf fun = bar;
23     f(fun); // #1, OK
24     g(fun); // #2, OK
25 }
```

#1 和 #2 处，分别调用了两个函数，参数为另一个函数，根据该条规则，ADL 得以查找到 A::f() 与 B::g()。

第二点，若实参类型是一个类，那么 ADL 会从该类或其父类的最内层命名空间进行查找。

例子如下：

```

1 namespace A {
2     // S2
3     struct Base {};
4 }
5
6 namespace M {
7     // S3 not works!
8     namespace B {
9         // S1
10        struct Derived : A::Base {};
11    }
12 }
13
14 int main() {
15     M::B::Derived d;
16     f(d); // #1
17 }
```

此处，若要通过 ADL 找到 f() 的定义，可以将其声明放在 S1 或 S2 处。

第三点，若实参类型是一个类模板，那么 ADL 会在特化类的模板参数类型的命名空间下进行查找；若实参类型包含模板模板参数，那么 ADL 还会在模板模板参数类型的命名空间下查找。

例子如下：

```

1 namespace C {
2     struct Final {};
3     void g(...) {
4         std::cout << "g found by ADL\n";
```

```

5         }
6     };
7
8 namespace B {
9     template <typename T>
10    struct Temtem {};
11
12   struct Bar {};
13   void f(...) {
14       std::cout << "f found by ADL\n";
15   }
16 }
17
18 namespace A {
19     template <typename T>
20     struct Foo {};
21 }
22
23 int main() {
24     // class template arguments
25     A::Foo<B::Bar> foo;
26     f(foo); // OK
27
28     // template template arguments
29     A::Foo<B::Temtem<C::Final>> a;
30     g(a); // OK
31
32 }

```

代码一目了然，不多解释。

第四点，当使用别名时，ADL 会无效，因为名称并不是一个函数调用。

看这个例子：

```

1 // Example from ISO C++
2
3 typedef int f;
4 namespace N {
5     struct A {
6         friend void f(A&);
7         operator int();
8         void g(A a) {
9             int i = f(a); // #1
10        }
11    };

```

```
12 }
```

注意 #1 处，并不会应用 ADL 来查询函数 f，因为它其实是 int，相当于调用 int(a)。

说完了这四点规则，下面来稍微说点 ADL 二段式相关的内容。

看下面这个例子：

```
1 namespace mylib {
2
3     struct S {};
4
5     void swap(S&, S&) {}
6
7     void play() {
8         using std::swap;
9
10        S s1, s2;
11        swap(s1, s2); // OK, found by Unqualified Name Lookup
12
13        int a1, a2;
14        swap(a1, a2); // OK, found by using declaration
15    }
16 }
```

然后，你要在某个地方调用自己提供的这个定制函数，此处是 play() 当中。

但是调用的地方，你需要的 swap() 可能不只是定制函数，还包含标准中的版本。因此，为了保证调用形式的唯一性，调用被分成了两步。

- 使用 using declaration
- 使用 swap()

这样一来，不同的调用就可以被自动查找到对应的版本上。然而，只要稍微改变下调用形式，代码就会出错：

```
1 namespace mylib {
2
3     struct S {};
4
5     void swap(S&, S&) {} // #1
6
7     void play() {
8         using namespace std;
9
10        S s1, s2;
11        swap(s1, s2); // OK, found by Unqualified Name Lookup
12
13        int a1, a2;
```

```

14     swap(a1, a2); // Error
15 }
16 }
```

这里将 using declaration 写成了 using directive，为什么就出错了？

其实，前者将 std::swap() 直接引入到了局部作用域，后者却将它引入了与最近的命名空间同等的作用域。根据前面讲过的准则：根据作用域查找顺序，当 Name Lookup 在某个作用域找到声明之后，便会停止查找。编译器查找到了 #1 处的定制函数，就立即停止，因此通过 using directive 引入的 std::swap() 实际上并没有被 Name Lookup 查找到。

这个细微的差异很难发现，标准在早期就犯了这个错误，因此 STL 中的许多实现存在不少问题，但由于 ABI 问题，又无法直接修复。这也是 C++20 引入 CPOs 的原因，STL2 Ranges 的设计就采用了这种新的定制点方式，以避免这个问题。

在这之前，标准发明了另一种方式来解决这个问题，称为 Hidden friends。

```

1 namespace mylib {
2
3     struct S {
4         // Hidden friends
5         friend void swap(S&, S&) {}
6     };
7
8     void play() {
9         using namespace std;
10
11         S s1, s2;
12         swap(s1, s2); // OK, found by ADL
13
14         int a1, a2;
15         swap(a1, a2); // OK
16     }
17 }
```

就是将定制函数定义为友元版本，放在类的内部。此时将不会再出现名称被隐藏的问题，这个函数只能被 ADL 找到。

Hidden friends 的写法在 STL 中存在不少，想必大家曾经也不知不觉中使用过。

好，更多关于定制点的内容本文不再涉及，下面进行另一个内容。

1.2.3 Template Name Lookup

以上两节 Name Lookup 内容只涉及零星关于模板的名称查找，本节专门讲解这部分查找，它们还是属于前两节的归类。

首先要说的是对于 typename 的使用，在模板当中声明一些类型，有些地方并不假设其为类型，此时只有在前面添加 typename，Name Lookup 才视其为类型。

不过自 C++20 之后，需要添加 typename 的地方已越来越少，已专门写过文章，请参考：新简化！typename 在 C++20 不再必要。

其次，介绍一个非常重要的概念，「独立名称」与「依赖名称」。

什么意思呢？看一个例子。

```

1 // Example from ISO C++
2
3 int j;
4
5 template <class T>
6 struct X {
7     void f(T t, int i, char* p) {
8         t = i; // #1
9         p = i; // #2
10        p = j; // #3
11    }
12 };

```

在 Name Lookup 阶段，模板还没有实例化，因此此时的模板参数都是未知的。对于依赖模板参数的名称，就称其为「依赖名称」，反之则为「独立名称」。

依赖名称，由于 Name Lookup 阶段还未知，因此对其查找和诊断要晚一个阶段，到模板实例化阶段。

独立名称，其有效性则在模板实例化之前，比如 #2 和 #3，它们诊断就比较早。这样，一旦发现错误，就不必再继续向下编译，节省编译时间。

查找阶段的变化对 Name Lookup 存在影响，看如下代码：

```

1 // Example from ISO C++
2
3 void f(char);
4
5 template <class T>
6 void g(T t) {
7     f(1);      // non-dependent
8     f(T(1)); // dependent
9     f(t);      // dependent
10    dd++;     // non-dependent
11 }
12
13 enum E { e };
14 void f(E);
15
16 double dd;
17 void h() {
18     g(e);    // calls f(char),f(E),f(E)
19     g('a'); // calls f(char),f(char),f(char)
20 }

```

在 h() 里面有两处对于 g() 的调用，而 g() 是个函数模板，于是其中的名称查找时间并不相同。

f(char) 是在 g() 之前定义的，而 f(E) 是在之后定义的，按照普通函数的 Name Lookup，理应是找不到 f(E) 的定义的。

但因为存在独立名称和依赖名称，于是独立名称会先行查找，如 f(1) 和 dd++，而变量 dd 也是在 g() 之后定义的，所以无法找到名称，dd++ 编译失败。对于依赖名称，如 f(T(1)) 和 f(t)，它们则是在模板实例化之后才进行查找，因此可以查找到 f(E)。

一言以蔽之，即使把依赖名称的定义放在调用函数之后，由于其查找实际上发生于实例化之后，故也可成功找到。

事实上，存在术语专门表示此种查找方式，称为 Two-phase Name Lookup（二段名称查找），在下节还会进一步讨论。

接着来看一个关于类外模板定义的查找规则。

看如下代码：

```

1 // Example from ISO C++
2
3 template <class T>
4 struct A {
5     struct B {};
6     typedef void C;
7     void f();
8     template<class U> void g(U);
9 };
10
11 template <class B>
12 void A<B>::f() {
13     B b; // #1
14 }
15
16 template <class B>
17 template <class C>
18 void A<B>::g(C) {
19     B b; // #2
20     C c; // #3
21 }
```

思考一下，#1,#2,#3 分别分别查找到的是哪个名称？（这个代码只有 clang 支持）

实际上，#1 和 #2 最终查找到的都是 A::B，而 #3 却是模板参数 C。

注意第 16-17 行出现的两个模板，它们并不能合并成一个，外层模板指的是类模板，而内层模板指的是函数模板。

因此，规则其实是：对于类外模板定义，如果成员不是类模板或函数模板，则类模板的成员名称会隐藏类外定义的模板参数；否则模板参数获胜。

而如果类模板位于一个命名空间之内，要在命名空间之外定义该类模板的成员，规则又不相同。

```

1 // Example from ISO C++
2
```

```

3  namespace N {
4      class C {};
5      template <class T> class B {
6          void f(T);
7      };
8  }
9
10 template <class C>
11 void N::B<C>::f(C) {
12     C b; // #1
13 }
```

此处，#1 处的 C 查找到的是模板参数。

如果是继承，那么也会隐藏模板参数，代码如下：

```

1 // Example from ISO C++
2
3 struct A {
4     struct B {};
5     int a;
6     int Y;
7 };
8
9 template <class B, class a>
10 struct X : A {
11     B b; // A::B
12     a b; // A::a, error, not a type name
13 };
```

这里，最终查找的都是父类当中的名称，模板参数被隐藏。

然而，如果父类是个依赖名称，由于名称查找于模板实例化之前，所以父类当中的名称不会被考虑，代码如下：

```

1 // Example from ISO C++
2
3 typedef double A;
4 template <class T>
5 struct B {
6     typedef int A;
7 };
8
9 template <class T>
10 struct X : B<T> {
11     A a; // double
12 };
```

这里，最终 X::A 的类型为 double，这是识别为独立名称并使用 Unqualified Name Lookup 查找到的。若要访问 B::A，那么声明改为 B::A a; 即可，这样就变为了依赖名称，且采用 Qualified Name Lookup 进行查找。

最后，说说多继承中包含依赖名称的规则。

还是看一个例子：

```

1 // Example from ISO C++
2
3 struct A {
4     int m;
5 };
6
7 struct B {
8     int m;
9 };
10
11 template <class T>
12 struct C : A, T {
13     int f() { return this->m; } // #1
14     int g() { return m; }       // #2
15 };
16
17 template int C<B>::f(); // ambiguous!
18 template int C<B>::g(); // OK

```

此处，多重继承包含依赖名称，名称查找方式并不相同。

对于 #1，使用 Qualified Name Lookup 进行查找，查询发生于模板实例化，于是存在两个实例，出现 ambiguous。

而对于 #2，使用 Unqualified Name Lookup 进行查找，此时相当于是独立名称查找，查找到的只有 A::m，所以不会出现错误。

1.2.4 Two-phase Name Lookup

因为模板才产生了独立名称与依赖名称的概念，依赖名称的查找需要等到模板实例化之后，这就是上节提到的二段名称查找。

依赖名称的存在导致 Unqualified Name Lookup 失效，此时，只有使用 Qualified Name Lookup 才能成功查找到其名称。

举个非常常见的例子：

```

1 struct Base {
2     // non-dependent name
3     void f() {
4         std::cout << "Base class\n";
5     }
6 };

```

```

7
8 struct Derived : Base {
9     // non-dependent name
10    void h() {
11        std::cout << "Derived class\n";
12        f(); // OK
13    }
14 };
15
16
17 int main() {
18     Derived d;
19     d.h();
20 }
21
22 // Outputs:
23 // Derived class
24 // Base class

```

这里，`f()` 和 `h()` 都是独立名称，因此能够通过 Unqualified Name Lookup 成功查找到名称，程序一切正常。

然而，把上述代码改成模板代码，情况就大不相同了。

```

1 template <typename T>
2 struct Base {
3     void f() {
4         std::cout << "Base class\n";
5     }
6 };
7
8 template <typename T>
9 struct Derived : Base<T> {
10    void h() {
11        std::cout << "Derived class\n";
12        f(); // error: use of undeclared identifier 'f'
13    }
14 };
15
16
17 int main() {
18     Derived<int> d;
19     d.h();
20 }

```

此时，代码已经无法编译通过了。

为什么呢？当编译器进行 Name Lookup 时，发现 `f()` 是一个独立名称，于是在模板定义之时就开始查找，然而很可惜，没有查找到任何结果，于是出现未定义的错误。

那么它为何不在基类当中查找呢？这是因为它的查找发生在第一阶段的 Name Lookup，此时模板还没有实例化，编译器不能草率地在基类中查找，这可能导致查找到错误的名称。

更进一步的原因在于，模板类支持特化和偏特化，比如我们再添加这样的代码：

```

1 template <>
2 struct Base<char> {
3     void f() {
4         std::cout << "Base<char> class\n";
5     }
6 };

```

若是草率地查找基类中的名称，那么查找到的将不是特化类当中的名称，查找出错。所以，在该阶段编译器不会在基类中查找名称。

那么，如何解决这个问题呢？

有两种办法，代码如下：

```

1 template <typename T>
2 struct Derived : Base<T> {
3     void h() {
4         std::cout << "Derived class\n";
5         this->f();      // method 1
6         Base<T>::f(); // method 2
7     }
8 };

```

这样一来，编译器就能够成功查找到名称。

原理是这样的：通过这两种方式，就可以告诉编译器该名称是依赖名称，必须等到模板实例化之后才能进行查找，届时将使用 Qualified Name Lookup 进行查找。这就是二段名称查找的必要性。在调用类函数模板时依旧存在上述问题，一个典型的例子：

```

1 struct S {
2     template <typename T>
3     static void f() {
4         std::cout << "f";
5     }
6 };
7
8 template <typename T>
9 void g(T* p) {
10    T::f<void>();           // #1 error!
11    T::template f<void>();   // #2 OK
12 }
13
14 int main() {

```

```

15     S s;
16     g(&s);
17 }
```

此处，由于 `f()` 是一个函数模板，#1 的名称查找将以失败告终。

因为它是一个依赖名称，编译器只假设名称是一个标识符（比如变量名、成员函数名），并不会认为它们是类型或函数模板。

原因如前面所说，由于模板特化和偏特化的存在，草率地假设会导致名称查找错误。此时，就需要显式地告诉编译器它们是一个类型或是函数模板，告诉编译器如何解析。

这也是需要对类型使用 `typename` 的原因，而对于函数模板，则如 #2 那样添加一个 `template`，这样就可以告诉编译器这是一个函数模板，`<>` 当中的名称于是被解析为模板参数。

#1 失败的原因也显而易见，编译器将 `f()` 当成了成员函数，将 `<>` 解析为了比较符号，从而导致编译失败。

至此，关于 Name Lookup 的内容就全部结束了，下面进入重载决议流程的第二阶段——模板处理。

1.3 Function Templates Handling

Name Lookup 查找的名称若是包含函数模板，那么下一步就需要将这些函数模板实例化。

模板实例化有两个步骤，第一个步骤是 Template Argument Deduction，对模板参数进行推导；第二个步骤是 Template Argument Substitution，使用推导出来的类型对模板参数进行替换。

下面两节分别介绍模板参数推导与替换的细节。

1.3.1 Template Argument Deduction

模板参数本身是抽象的类型，并不真正存在，因此需要根据调用的实参进行推导，从而将类型具体化。

TAD 就描述了如何进行推导，规则是怎样的。

先来看一个简单的例子，感受一下基本规则。

```

1 // Example from ISO C++
2
3 template <class T, class U = double>
4 void f(T t = 0, U u = 0) {
5 }
6
7
8 int main() {
9     f(1, 'c');           // f<int, char>(1, 'c');
10    f(1);               // f<int, double>(1, 0)
11    f();                // error: T cannot be deduced
12    f<int>();          // f<int, double>(0, 0)
13    f<int, char>();    // f<int, char>(0, 0)
14 }
```

调用的实参是什么类型，模板参数就自动推导为所调用的类型。如果模板参数具有默认实参，那么可以从其推导，也可以显式指定模板参数，但若没有任何参数，则不具备推导上下文，推导失败。

这里存在令许多人都比较迷惑的一点，有些时候推导的参数并不与调用实参相同。

比如：

```

1 template <class T>
2 void f(T t) {}
3
4 int main() {
5     const int i = 1;
6     f(i); // T deduced as int, f<int>(int)
7 }
```

这里实参类型是 `const int`，但最后推导的却是 `int`。

这是因为，推导之时，所有的 top-level 修饰符都会被忽略，此处的 `const` 为 top-level `const`，于是 `const` 被丢弃。本质上，其实是因为传递过去的参数变量实际上是新创建的拷贝变量，原有的修饰符不应该影响拷贝之后的变量。

那么，此时如何让编译器推导出你想要的类型呢？

第一种办法，显示指定模板参数类型。

```
f<const int>(i); // OK, f<const int>(const int)
```

第二种办法，将模板参数声明改为引用或指针类型。

```

template <class T>
void f(T& t) {}

f(i); // OK, f<const int>(const int&)
```

为什么改为引用或指针就可以推导出带 `const` 的类型呢？

这是因为此时变量不再是拷贝的，它们访问的依旧是实参的内存区域，如果忽略掉 `const`，它们将能够修改 `const` 变量，这会导致语义错误。

因此，如果你写出这样的代码，推导将会出错：

```

1 template <class T>
2 void f(T t1, T* t2) {}
3
4 int main() {
5     const int i = 1;
6     f(i, &i); // Error, T deduced as both int and const int
7 }
```

因为根据第一个实参，`T` 被推导为 `int`，而根据第二个实参，`T` 又被推导为 `const int`，于是编译失败。

若你显示指定参数，那么将可以消除此错误，代码如下：

```

1 template <class T>
2 void f(T t1, T* t2) { }
3
4 int main() {
5     const int i = 1;
6     f<const int>(i, &i); // OK, T has const int type
7 }
```

此时，T 的类型只为 const int，冲突消除，于是编译成功。

下面介绍可变参数模板的推导规则。

看如下例子：

```

1 template <class T, class... Ts>
2 void f(T, Ts...) {
3 }
4
5 template <class T, class... Ts>
6 void g(Ts..., T) {
7 }
8
9
10 int main() {
11     f(1, 'c', .0);    // f<int, char, double>(int, char, double)
12     //g(1, 'c', .0); // error, Ts is not deduced
13 }
```

此处规则为：参数包必须放到参数定义列表的最末尾，TAD 才会进行推导。

但若是参数包作为类模板参数出现，则不必遵循此顺序也可以正常推导。

```

1 template <class...>
2 struct Tuple {};
3
4 template <class T, class... Ts>
5 void g(Tuple<Ts...>, T) {
6 }
7
8 g(Tuple<int>{}, .0); // OK, g<int, double>(Tuple<int>, double)
```

如果函数参数是一个派生类，其继承自类模板，类模板又采用递归继承，则推导实参为其直接基类的模板参数。示例如下：

```

1 // Example from ISO C++
2
3 template <class...> struct X;
4 template <> struct X<> {};
5 template <class T, class... Ts>
```

```

6  struct X<T, Ts...> : X<Ts...> {};
7  struct D : X<int> {};
8
9  template <class... Ts>
10 int f(const X<Ts...>&) {
11     return {};
12 }
13
14
15 int main() {
16     int x = f(D()); // deduced as f<int>, not f<>
17 }
```

这里，最终推导出来的类型为 f，而非 f<>。

下面介绍 forwarding reference 的推导规则。

对于 forwarding reference，如果实参为左值，则模板参数推导为左值引用。看一个不错的例子：

```

1 // Example from ISO C++
2
3 template <class T> int f(T&& t);
4 template <class T> int g(const T&&);

5
6 int main() {
7     int i = 1;
8     //int n1 = f(i); // #1, f<int&>(int&)
9     //int n2 = f(0); // #2, f<int>(int&&);
10    int n3 = g(i); // #3, g<int>(const int&&)
11                      // error: bind an rvalue reference to an lvalue
12 }
```

此处，f() 的参数为 forwarding reference，g() 的参数为右值引用。

因此，当实参为左值时，f() 的模板参数被推导为 int&，g() 的模板参数则被推导为 int。而左值无法绑定到右值，于是编译出错。

再来看另一个例子：

```

1 // Example from ISO C++
2
3 template <class T>
4 struct A {
5     template <class U>
6     A(T&& t, U&& u, int*); // #1
7
8     A(T&&, int*); // #2
9 };
10
11 template <class T> A(T&&, int*) -> A<T>; // #3
```

对于 #1, U&& 为 forwarding reference, 而 T&& 并不是, 因为它不是函数模板参数。

于是, 当使用 #1 初始化对象时, 若第一个实参为左值, 则 T&& 被推导为右值引用。由于左值无法绑定到右值, 遂编译出错。但是第二个参数可以为左值, U 会被推导为左值引用, 次再施加引用折叠, 最终依旧为左值引用, 可以接收左值实参。

若要使类模板参数也变为 forwarding reference, 可以使用 CTAD, 如 #3 所示。此时, T&& 为 forwarding reference, 第一个实参为左值时, 就可以正常触发引用折叠。

1.3.2 Template Argument Substitution

TAD 告诉编译器如何推导模板参数类型, 紧随其后的就是使用推导出来的类型替换模板参数, 将模板实例化。

这两个步骤密不可分, 故在上节当中其实已经涉及了部分本节内容, 这里进一步扩展。

这里只讲三个重点。

第一点, 模板参数替换存在失败的可能性。

模板替换并不总是会成功的, 比如:

```

1 struct A { typedef int B; };

2

3 template <class T> void g(typename T::B*) // #1
4 template <class T> void g(T);           // #2
5
6 g<int>(0); // calls #2

```

Name Lookup 查找到了 #1 和 #2 的两个名称, 然后对它们进行模板参数替换。然而, 对于 #1 的参数替换并不能成功, 因为 int 不存在成员类型 B, 此时模板参数替换失败。

但是编译器并不会进行报错, 只是将其从重载集中移除。这个特性就是广为熟知的 SFINAE(Substitution Failure Is Not An Error), 后来大家发现该特性可以进一步利用起来, 为模板施加约束。

比如根据该原理可以实现一个 enable_if 工具, 用来约束模板。

```

1 namespace mylib {
2
3     template <bool, typename = void>
4     struct enable_if {};
5
6     template <typename T>
7     struct enable_if<true, T> {
8         using type = T;
9     };
10
11    template <bool C, typename T = void>
12    using enable_if_t = typename enable_if<C, T>::type;
13
14 } // namespace mylib
15
16

```

```

17 template <typename T, mylib::enable_if_t<std::same_as<T, double>, bool> = true>
18 void f() {
19     std::cout << "A\n";
20 }
21
22 template <typename T, mylib::enable_if_t<std::same_as<T, int>, bool> = true>
23 void f() {
24     std::cout << "int\n";
25 }
26
27 int main() {
28     f<double>(); // calls #1
29     f<int>(); // calls #2
30 }
```

`enable_if`早已加入了标准，这个的工具的原理就是利用模板替换失败的特性，将不符合条件的函数从重载集移除，从而实现正确的逻辑分派。

SFINAE 并非是专门针对类型约束而创造出来的，使用起来比较复杂，并不直观，已被 C++20 的 Concepts 取代。

第二点，关于 trailing return type 与 normal return type 的本质区别。

这二者的区别本质上就是 Name Lookup 的区别：normal return type 是按照从左到右的词法顺序进行查找并替换的，而 trailing return type 因为存在占位符，打乱了常规的词法顺序，这使得它们存在一些细微的差异。

比如一个简单的例子：

```

1 namespace N {
2     using X = int;
3     X f();
4 }
5
6 N::X N::f(); // normal return type
7 auto N::f() -> X; // trailing return type
```

根据前面讲述的 Qualified Name Lookup 规则，normal return type 的返回值必须使用 `N::X`，否则将在全局查找。而 trailing return type 由于词法顺序不同，可以省略这个命名空间。

当然 trailing return type 也并非总是比 normal return type 使用起来更好，看如下例子：

```

1 // Example from ISO C++
2
3 template <class T> struct A { using X = typename T::X; };
4
5 // normal return type
6 template <class T> typename T::X f(typename A<T>::X);
7 template <class T> void f(...);
8
```

```

9 // trailing return type
10 template <class T> auto g(typename A<T>::X) -> typename T::X;
11 template <class T> void g(...);
12
13
14 int main() {
15     f<int>(0); // #1 OK
16     g<int>(0); // #2 Error
17 }
```

通常来说，这两种返回类型只是形式上的差异，是可以等价使用的，但此处却有着细微而本质的区别。

#1 都能成功调用，为什么改了个返回形式，#2 就编译出错了？

这是因为：

在模板参数替换的时候，normal return type 遵循从左向右的词法顺序，当它尝试替换 `T::X`，发现实参类型 `int` 并没有成员 `X`，于是依据 SFINAE，该名称被舍弃。然后，编译器发现通用版本的名称可以成功替换，于是编译成功。

而 #2 在模板参数替换的时候，首先跳过 `auto` 占位符，开始替换函数参数。当它尝试使用 `int` 替换 `A::X` 的时候，发现无法替换。但是 `A::X` 并不会触发 SFINAE，而是产生 hard error，于是编译失败。

简单来说，此处，normal return type 在触发 hard error 之前就触发了 SFINAE，所以可以成功编译。

第三点，forwarding reference 的模板参数替换要点。

看一个上周我在群内分享的一个例子：

```

1 template <class T>
2 struct S {
3     static void g(T&& t) {}
4     static void g(const T& t) {}
5 };
6
7
8 template <class T> void f(T&& t) {}
9 template <class T> void f(const T& t) {}

10
11 int main() {
12     int i = 1;
13
14     f<int&>(i);    // #1 OK
15     S<int&>::g(i); // #2 Error
16 }
```

为什么 #1 可以通过编译，而 #2 却不可以呢？

首先来分析 #2，编译失败其实显而易见。

由于调用显式指定了模板参数，所以其实并没有参数推导，`int&` 用于替换模板参数。对于 `T&&`，替换为 `int&&&`，1 折叠后变为 `int&`；对于 `const T&`，替换为 `const (int&)&`，等价于 `int& const&`，而 C++ 不支持 top-level reference，`int& const` 声明本身就是非法的，所以 `const` 被抛弃，剩下 `int&&`，折叠为 `int&`。

于是重复定义，编译错误。

而对于 #1，它包含两个函数模板。若是同时替换，那么它们自然也会编译失败。但是，根据 4.3 将要介绍的规则：如果都是函数模板，那么更特殊的函数模板胜出。`const T&` 比 `T&&` 更加特殊，因此 `f(T&&)` 最终被移除，只存在 `f(const T&)` 替换之后的函数，没有错误也是理所当然。

1.4 Overload Resolution

经过 Name Lookup 和 Template Handling 两个阶段，编译器搜索到了所有相关重载函数名称，这些函数就称为 candidate functions（候选函数）。

前文提到过，Name Lookup 仅仅只是进行名称查找，并不会检查这些函数的有效性。因此，candidate functions 只是「一级筛选」的结果。

重载决议，就是要在一级筛选的结果之上，选择出最佳的那个匹配函数。

比如：参数个数是否匹配？实参和形参的类型是否相同？类型是否可以转换？这些都属于筛选准则。

因此，这一步也可以称之为「二级筛选」。根据筛选准则，剔除掉无效函数，剩下的结果就称为 viable functions（可行函数）。

存在 viable functions，就表示已经找到可以调用的声明了。但是，这个函数可能存在多个可用版本，此时，就需要进行「终极筛选」，选出最佳的匹配函数，即 best viable function。终极筛选在标准中也称为 Tiebreakers（决胜局）。

终极筛选之后，如果只会留下一个函数，这个函数就是最终被调用的函数，重载决议成功；否则的话重载决议失败，程序错误。

接下来，将从一级筛选开始，以一个完整的例子，为大家串起整个流程，顺便加深对前面各节内容的理解。

1.4.1 Candidate functions

一级筛选的结果是由 Name Lookup 查找出来的，包含成员和非成员函数。

对于成员函数，它的第一个参数是一个额外的隐式对象参数，一般来说就是 `this` 指针。

对于静态成员函数，大家都知道它没有 `this` 指针，然而事实上它也存在一个额外的隐式对象参数。究其原因，就是为了重载决议可以正常运行。

可以看如下例子来进行理解。

```

1  struct S {
2      void f(long) {
3          std::cout << "member version\n";
4      }
5      static void f(int) {
6          std::cout << "static member version\n";
7      }
8  };

```

```

9
10 int main() {
11     S s;
12     s.f(1); // calls static member version
13 }
```

此时，这两个成员函数实际上为：

```
f(S&, long); // member version
f(implicit object parameter, int); // static member version
```

如果静态成员函数没有这个额外的隐式对象，那么其一，将可以定义一个参数完全相同的非静态成员；其二，重载决议将无法选择最佳的那个匹配函数（此处 long 需要转换，不是最佳匹配函数）。

静态成员的这个隐式对象参数被定义为可以匹配任何参数，仅仅用于在重载决议阶段保证操作的一致性。

对于非成员函数，则可以直接通过 Unqualified Name Lookup 和 Qualified Name Lookup 找到。同时，模板实例化后也会产生成员或非成员函数，除了有些因为模板替换失败被移除，剩下的名称共同组成了 candidate functions。

1.4.2 Viable functions

二级筛选要在 candidate functions 的基础上，通过一些筛选准则来剔除不符合要求的函数，留下的是 viable functions。

筛选准则主要看两个方面，一个是看参数匹配程度，另一个是看约束满足程度。

约束满足就是看是否满足 Concepts，这是 C++20 之后新增的一项检查。

具体的检查流程如下所述。

第一步，看参数个数是否匹配。

假设实参个数为 N，形参个数为 M，则存在三种比较情况。

如果 N 等于 M，这种属于个数完全匹配，此类函数将被留下。

如果 N 小于 M，此时就需要看 candidate functions 是否存在默认参数，如果不存在，此类函数被淘汰。

如果 N 大于 M，此时就需要看 candidate functions 是否存在可变参数，如果不存在，此类函数被淘汰。

第二步，是否满足约束。

第一轮淘汰过后，剩下的函数如果存在 Concepts 约束，这些约束应该被满足。如果不满足，此类函数被淘汰。

第三步，看参数是否匹配。

实参类型可能和 candidate functions 完全匹配，也可能不完全匹配，此时这些参数需要存在隐式转换序列。可以是标准转换，也可以是用户自定义转换，也可以是省略操作符转换。

这三步过后，留下的函数就称为 viable functions，它们都有望成为最佳匹配函数。

1.4.3 Tiebreakers

终极筛选也称为决胜局，重载决议的最后一步，将进行更加严格的匹配。

第一，它会看参数的匹配程度。

如前所述，实参类型与 viable functions 可能完全匹配，也可能需要转换，此时就存在更优的匹配选项。

C++ 的类型转换有三种形式，标准转换、自定义转换和省略操作符转换。

标准转换比自定义转换更好，自定义转换比省略操作符转换更好。

对于标准转换，可以看下表。

Conversion	Category	Rank	
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		
Array-to-pointer conversion			
Function-to-pointer conversion			
Qualification conversions	Qualification Adjustment		
Function pointer conversion			
Integral promotions	Promotion	Promotion	
Floating-point promotion			
Integral conversions	Conversion	Conversion	
Floating-point conversions			
Floating-integral conversions			
Pointer conversions			
Pointer-to-member conversions			
Boolean conversions			

它们的匹配优先级也是自上往下的，即 Exact Match 比 Promotion 更好，Promotion 比 Conversion 更好，可以理解为完全匹配、次级匹配和低级匹配。

看一个简单的例子：

```

1 void f(int);
2 void f(char);
3
4 int main() {
5     f(1); // f(int) wins
6 }
```

此时，viable functions 就有两个。而实参类型为 int，f(int) 不需要转换，而 f(char) 需要将 int 转换为 char，因此前者胜出。

如果实参类型为 double，由于 double 转换为 int 和 char 属于相同等级，因此谁也不比谁好，产生 ambiguous。

再来看一个例子：

```

1 // Example from ISO C++
2
3 void f(const int*, short);
4 void f(int*, int);
5
6 int main() {
7     int i;
8     short s = 0;
```

```

9     f(&i, s);      // #1 Error, ambiguous
10    f(&i, 1L);    // #2 OK, f(int*, int) wins
11    f(&i, 'c');   // #3 OK, f(int*, int) wins
12 }

```

这里存在两个 viable functions，存在一场决胜局。

#1 处调用，第一个实参类型为 int，第二个实参类型为 short。对于前者来说，f(int, int) 是更好的选择，而对于后者来说，f(const int*, short) 才是更好的选择。此时将难分胜负，因此产生 ambiguous。

#2 处调用，第二个实参类型为 long，打成平局，但 f(int*, int) 在第一个实参匹配中胜出，因此最终被调用。

#3 处调用，第二个实参类型为 char，char 转换为 int 比转换为 short 更好，因此 f(int*, int) 依旧胜出。

对于派生类，则子类向直接基类转换是更好的选择。

```

1 struct A {};
2 struct B : A {};
3 struct C : B {};
4
5 void f(A*) {
6     std::cout << "A*";
7 }
8 void f(B*) {
9     std::cout << "B*";
10}
11
12 int main() {
13     C* pc;
14     f(pc); // f(B*) wins
15 }

```

这里，C 向 B 转换，比向 A 转换更好，所以 f(B*) 胜出。

最后再来看一个例子，包含三种形式的转换。

```

1 struct A {
2     operator int();
3 };
4
5 void f(A) {
6     std::cout << "standard conversion wins\n";
7 }
8
9 void f(int) {
10    std::cout << "user defined conversion wins\n";
11 }
12

```

```

13 void f(...) {
14     std::cout << "ellipsis conversion wins\n";
15 }
16
17 int main() {
18     A a;
19     f(a);
20 }
```

最终匹配的优先级是从上往下的，标准转换是优先选择，自定义转换次之，省略操作符转换最差。

第二，如果同时出现模板函数和非模板函数，则非模板函数胜出。

例子如下：

```

1 void f(int) {
2     std::cout << "f(int) wins\n";
3 }
4
5 template <class T>
6 void f(T) {
7     std::cout << "function templates wins\n";
8 }
9
10 int main() {
11     f(1); // calls f(int)
12 }
```

但若是非模板函数还需要参数转换，那么模板函数将胜出，因为模板函数可以完全匹配。

第三，如果都是函数模板，那么更特殊的模板函数胜出。

什么是更特殊的函数模板？其实指的就是更加具体的函数模板。越抽象的模板参数越通用，越具体的越特殊。举个例子，语言、汉语和普通话，语言可以表示汉语，汉语可以表示普通话，因此语言比汉语更抽象，汉语比普通话更抽象，普通话比汉语更特殊，汉语又比语言更特殊。

越抽象越通用，越具体越精确，越精确就越可能是实际的调用需求，因此更特殊的函数模板胜出。

比如在 3.2 节第三点提到的例子，`const T&` 为何比 `T&&` 更特殊呢？这是因为，若形参类型为 `T`，实参类型为 `const U`，则 `T` 可以推导为 `const U`，前者就可以表示后者。若是反过来，形参类型为 `const T`，实参类型为 `U`，此时就无法推导。因此 `const T&` 要更加特殊。

第四，如果都函数都带有约束，那么满足更多约束的获胜。

例子如下：

```

1 // Example from ISO C++
2
3 template<typename T> concept C1 = requires(T t) { --t; };
4 template<typename T> concept C2 = C1<T> && requires(T t) { *t; };
5
```

```

6  template<C1 T> void f(T);           // #1
7  template<C2 T> void f(T);           // #2
8  template<class T> void g(T);        // #3
9  template<C1 T> void g(T);           // #4

10
11 int main() {
12     f(0);      // selects #1
13     f((int*)0); // selects #2
14     g(true);   // selects #3 because C1<bool> is not satisfied
15     g(0);      // selects #4
16 }

```

第五，如果一个是模板构造函数，一个是非模板构造函数，那么非模板版本获胜。例子如下：

```

1 template <class T>
2 struct S {
3     S(T, T, int);                  // #1
4     template <class U> S(T, U, int); // #2
5 };
6
7 int main() {
8     // selects #1, generated from non-template constructor
9     S s(1, 2, 3);
10 }

```

究其原因，还是非模板构造函数更加特殊。

以上所列的规则都是比较常用的规则，更多规则大家可以参考 [cppreference](#)。

通过这些规则，就可以找出最佳匹配的那个函数。如果最后只剩下一个 viable function，那么它就是 best viable function。如果依旧存在多个函数，那么 ambiguous。

大家也许还不是特别清楚上述流程，那么接下来，我将以一个完整的例子来串起整个流程。

1.5 走一遍完整的流程

一个完整的示例，代码如下：

```

1 namespace N {
2     struct Base {};
3     struct Derived : Base { void foo(Base* s, char); }; // #1
4     void foo(Base* s, char);                         // #2
5     void foo(Derived* s, int, bool = true); // #3
6     void foo(Derived* s, short);                   // #4
7 }
8
9 struct S {

```

```

10     N::Derived* d;
11     S(N::Derived* deri) : d{deri} {}
12     operator N::Derived*() const { return d; }
13 };
14
15 void foo(S); // #5
16 template <class T> void foo(T* t, int c); // #6
17 void foo(...); // #7
18
19 int main() {
20     N::Derived d;
21     foo(&d, 'c'); // which one will be matched?
22 }
```

最终哪个函数能够胜出，让我们来逐步分析。

第一步，编译器会进行 Name Lookup，查找名称。如图1.1。

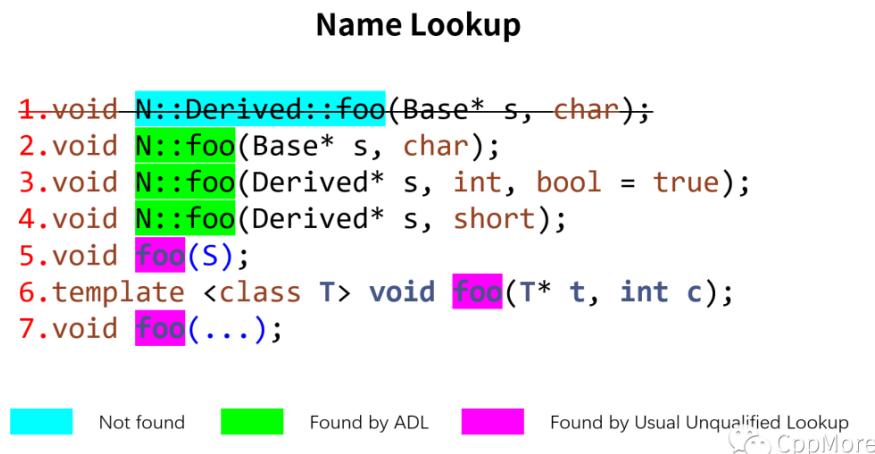


图 1.1: Name Lookup

可以看到，代码中一共有七个重载函数，但是只会被查找到六个。因为 `foo(&d, 'c')` 调用时没有添加任何作用域限定符，所以编译器不会使用 Qualified Name Lookup 进行查找。

在查找到的六个名称当中，其中有三个是通过 ADL 查找到的，还有三个是通过 Usual Unqualified Lookup 查找到的。

第二步，编译器发现其中包含函数模板，于是进行 Template Handling。如图1.2。

首先，编译器根据调用实参，通过 Template Argument Deduction 推导出实参类型，实参类型如上图 A1 和 A2 所示。

接着，编译器分析函数模板中包含的模板参数，其中 P1 为模板参数。于是，需要进行 Template Argument Substitution，将 P1 替换为实参类型，T 被替换为 N::Derived。

如果模板替换失败，根据 SFINAE，这些函数模板将被移除。

最后，替换完成的函数就和其他的函数一样，它们共同构成了 candidate functions，一级筛选到此结束。如图1.3。

第三步，编译器正式进入重载决议阶段，比较 candidate functions，选择最佳匹配函数。

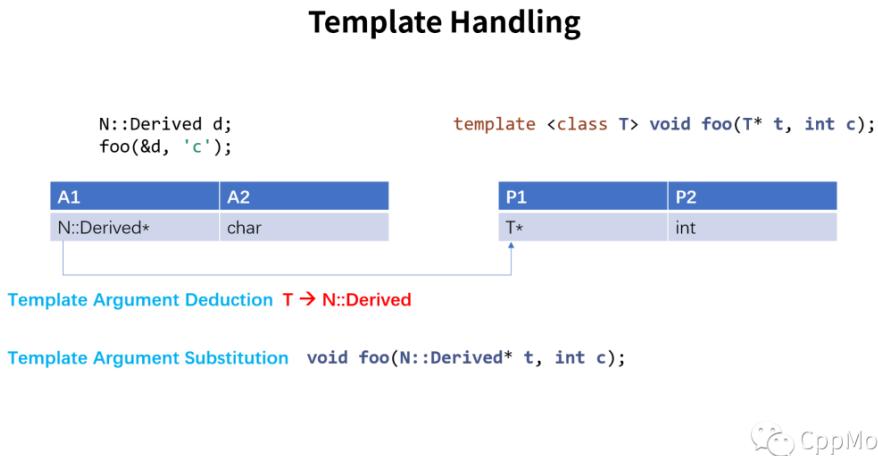


图 1.2: Template Handling

Candidate Functions

```

1. void N::foo(Base* s, char);
2. void N::foo(Derived* s, int, bool = true);
3. void N::foo(Derived* s, short);
4. void foo(S);
5. void foo(N::Derived* t, int c);
6. void foo(...);

```

© CppMore

图 1.3: Candidate Functions

首先，进行二级筛选，筛选掉明显不符合的候选者。

调用参数为 2 个，而第 4 个候选者只有 1 个参数，被踢出局；第 2 个候选者具有 3 个参数，但是它的第三个参数设有缺省值，因此依旧被留下。

此外，这些候选函数也没有任何约束，因此在这一局只剔除了一个函数，剩下的函数就称为 *viable functions*。如图 1.4。

*viable functions*之所以称为可行函数，就是因为它们其实都可以作为最终的调用函数，只是谁更好而已。

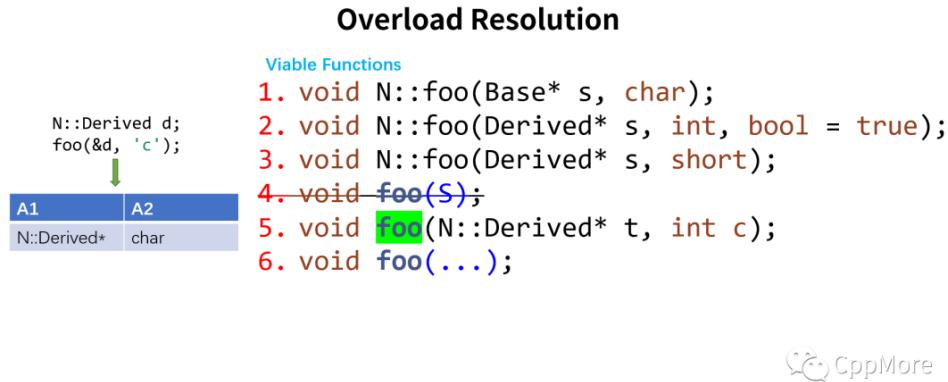
其次，进行终级筛选，即决胜局。在此阶段，需要比较参数的匹配程序。

对于派生类，完全匹配比直接基类好，直接基类比间接基类好，因此第 1 个候选者被踢出局。

第 6 个候选者为省略操作符，它将永远是最后才会被考虑的对象，也是最差的匹配对象。于是，2、3、5 进行决战。

它们的第一个参数都是完全匹配，因此看第二个参数。`char` 转换为 `int` 比 `short` 更好，因此第 3 个候选者被踢出局。

剩下第 2、5 个候选者，第 2 个候选者虽然有三个参数，但因为有缺省值，所以并不影响，也不



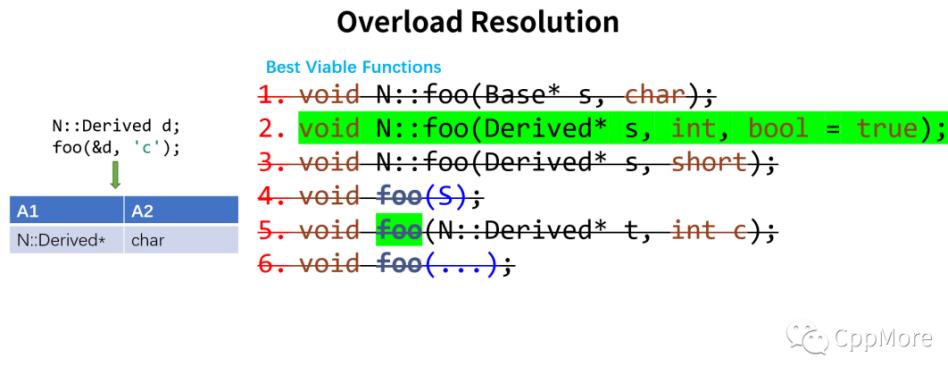
CppMore

图 1.4: Viable Functions

会被作为决胜因素，所以第 5 个候选者暂时还无法取胜。

然后，编译器发现第 2 个候选者为非模板函数，第 5 个候选者为模板函数。模板函数和非模板函数同时出现时，非模板函数胜出，于是第 5 个候选者被踢出局。

最后，只留下了第 2 个候选者，它成为了 best viable function，胜利者。如图1.5。



CppMore

图 1.5: Best Viable Functions

但是，大家可别以为竞选出胜利者就一定可以调用成功。事实上，它们只针对的是声明，如果函数没有定义，依旧会编译失败。

1.6 Name Mangling

重载函数的名称实际上是通过 Name Mangling 生成的新名称，大家如果去看编译后的汇编代码就能够看到这些名称。

像是 Compiler Explorer，它实际上是为了让你看着方便，显示的是优化后的名称，去掉勾选 Demangle identifiers 就能够看到实际函数名称。如图1.6。

那么接下来，就来介绍一下 Name Mangling 的实际手法。标准并没有规定具体实现方式，因此编译器的实际可能不尽相同，下面以 gcc 为例进行分析。

下面是使用 gcc 编译过后的一个例子。如图1.7。

如图所示，编译器为每个重载函数都生成一个新名称，新名称是绝对唯一的。

基本的规则如下图所示。如图1.8。

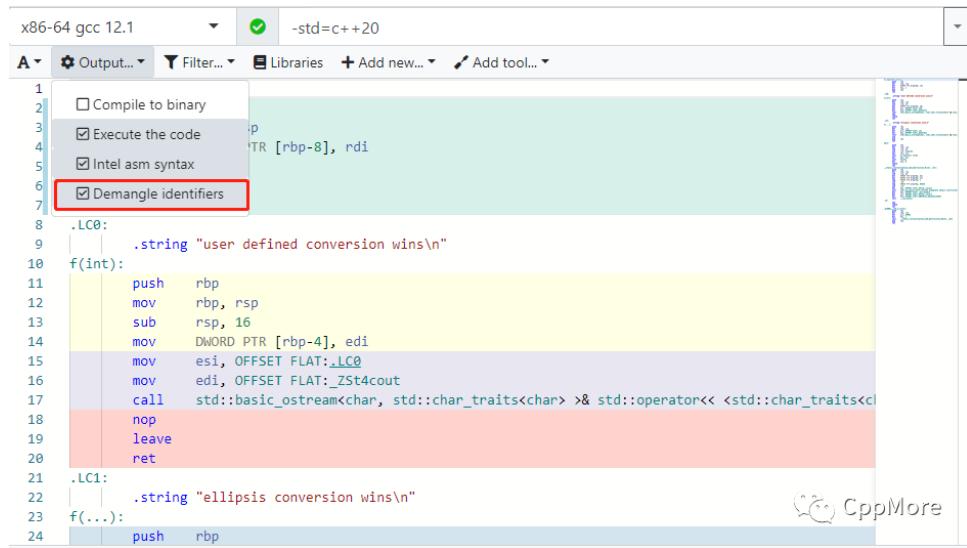


图 1.6: Compiler Explorer Demangle identifiers

Original Name	Mangled Name
void foo();	_Z3foov
void foo(int, char);	_Z3fooic
void foo(A, bool);	_Z3foo1Ab
int foo(float*);	_Z3fooPf
void foo(int A::*);	_Z3fooM1Ai

图 1.7: Mangled name

除了基本规则，还有很多比较复杂的规则，这里再举几个常见的名称。

```

1  namespace myNS {
2      struct myClass {
3          // mangles as _ZN4myNS7myClass6myFuncEv
4          void myFunc() {}
5      };
6
7      // mangles as _ZN4myNS3fooENS_7myClassE
8      void foo(myClass) {}
9  }
10
11 template <class T>
12 void foo(T, int) {}
13
14 // mangles as _Z3fooIfEvT_i
15 template void foo(float, int);

```

规则不难理解，大家可以自己找下规律。其中，I/E 中间的是模板参数，T_ 表示第 1 个模板参数。

Type Name	Encoding Name
int	i
char	c
float	f
void	v
long	l
double	d
array or pointer	P
user defined types	类型长度+类型名称
Pointer-to-member types	M

图 1.8: CppMore

图 1.8: Mangled rules

由于 C 语言没有重载函数，所以它也没有 Mangling 操作。如果你使用混合编译，即某些文件使用 C 编译，某些文件使用 C++ 编译，就会产生链接错误。

举个例子，有如下代码：

```

1 // lib.cpp
2 int myFunc(int a, int b) {
3     return a + b;
4 }
5
6 // main.cpp
7 #include <iostream>
8
9 int myFunc(int a, int b);
10
11 int main() {
12     std::cout << "The answer is " << myFunc(41, 1);
13 }
```

使用 C++ 编译并链接，结果如下图。如图1.9。

编译器在编译 main.cpp 时，发现其中存在一个未解析的引用 int myFunc(int a, int b);，于是在链接文件 lib.cpp 中找到了该定义。之所以能够找到该定义，是因为这两个文件都是使用 C++ 编译的，编译时 main.cpp 中的声明经过 Name Mangling 变为 _Z6myFuncii，实际查找的并不是 myFunc 这个名称。而 lib.cpp 中的名称也经过了 Name Mangling，因此能够链接成功。

但是，如果其中一个文件使用 C 进行编译，另一个使用 C++ 进行编译，链接时就会出现问题。如图1.10。

由于 main.cpp 是用 C++ 编译的，因此实际查找的名称为 _Z6myFuncii。而 lib.cpp 是用 C 编译的，并没有经过 Name Mangling，它的名称依旧为 myFunc，因此出现未定义的引用错误。

常用解法是使用一个 extern 关键字，告诉编译器这个函数来自 C，不要进行 Name Mangling。

```

1 // main.cpp
2 extern "C" int myFunc(int a, int b);
```

```

PS C:\projects\cpp\temp> ls

    目录: C:\projects\cpp\temp

Mode                LastWriteTime         Length Name
----                ——————              ———— ——
-a---        2022/9/4     16:17             50 lib.cpp
-a---        2022/9/4     16:19            119 main.cpp

PS C:\projects\cpp\temp> g++ -c lib.cpp
PS C:\projects\cpp\temp> g++ -c main.cpp
PS C:\projects\cpp\temp> g++ lib.o main.o -o main
PS C:\projects\cpp\temp> ./main
The answer is 42
PS C:\projects\cpp\temp>

```

图 1.9: Results

```

PS C:\projects\cpp\temp> gcc -c -x c lib.cpp
PS C:\projects\cpp\temp> g++ -c main.cpp
PS C:\projects\cpp\temp> g++ lib.o main.o -o main
C:/msys64/mingw64/bin/../lib/gcc/x86_64-w64-mingw32/11.2.0/../../../../x86_64-w64-mingw32/bin/ld.exe: main.o:main.cpp:(.text+0x37): undefined reference to 'myFunc(int, int)'
collect2.exe: error: ld returned 1 exit status
PS C:\projects\cpp\temp> |

```

图 1.10: 链接错误

```

3
4 int main() {
5     std::cout << "The answer is " << myFunc(41, 1);
6 }

```

如此一来，就可以解决这个问题。如图1.11。

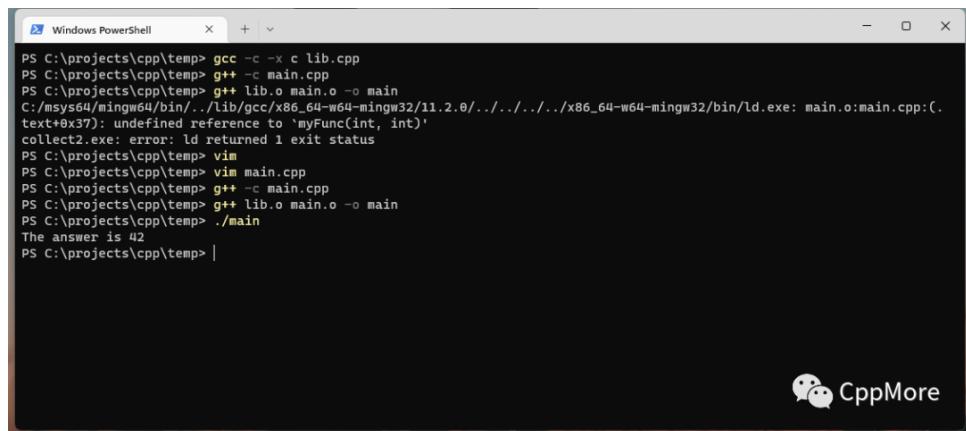
通常来说，可以使用预处理条件语句，分别提供 C 和 C++ 版本的代码，这样使用任何方式就都可以编译成功。

1.7 总结

本篇的内容相当之多，完整地包含了重载决议的整个流程。

能读到这里，相信大家已经收获满满，对整个流程已经有了清晰的认识。

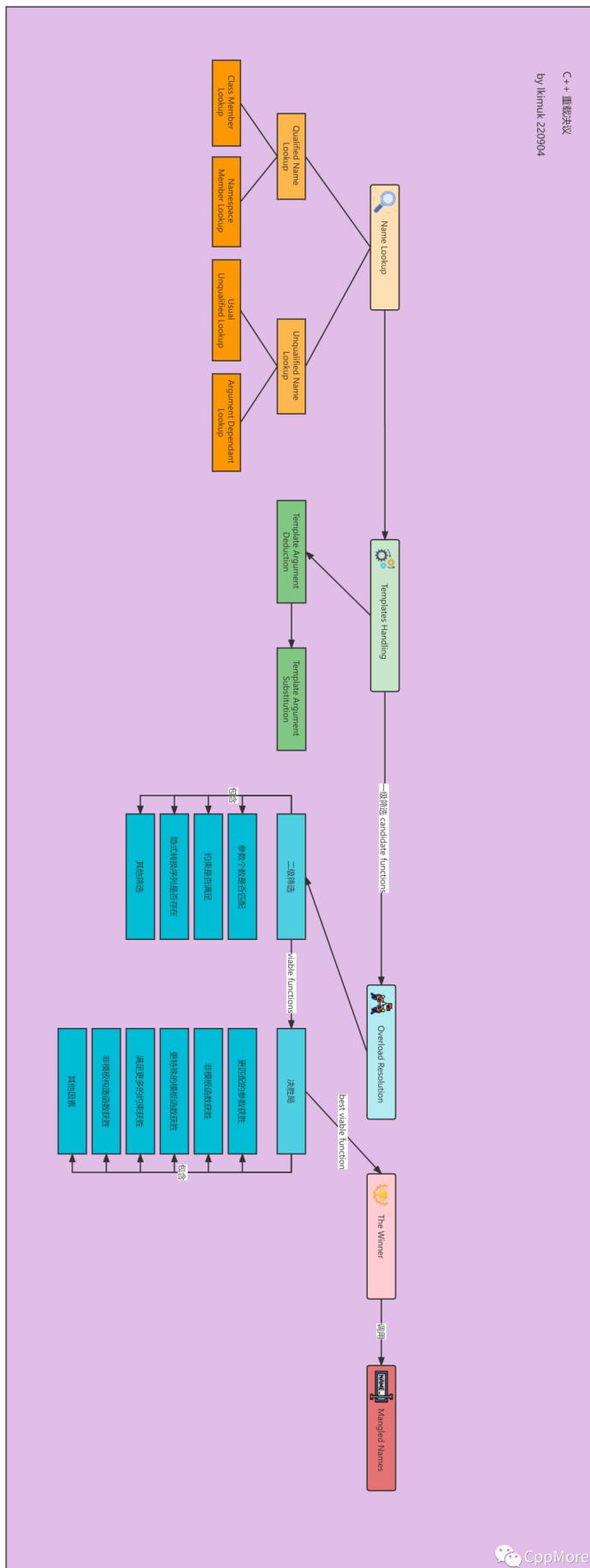
因此，这个总结算是一个课后作业，请大家对照下图回想本篇内容，如果对所有概念都十分清楚，那么恭喜你已经理解了重载决议！



The screenshot shows a Windows PowerShell window titled "Windows PowerShell". The command history is as follows:

```
PS C:\projects\cpp\temp> gcc -c -x c lib.cpp
PS C:\projects\cpp\temp> g++ -c main.cpp
PS C:\projects\cpp\temp> g++ lib.o main.o -o main
C:/msys64/mingw64/bin/../lib/gcc/x86_64-w64-mingw32/11.2.0/../../../../x86_64-w64-mingw32/bin/ld.exe: main.o:main.cpp:(.text+0x37): undefined reference to `myFunc(int, int)'
collect2.exe: error: ld returned 1 exit status
PS C:\projects\cpp\temp> vim
PS C:\projects\cpp\temp> vim main.cpp
PS C:\projects\cpp\temp> g++ -c main.cpp
PS C:\projects\cpp\temp> g++ lib.o main.o -o main
PS C:\projects\cpp\temp> ./main
The answer is 42
PS C:\projects\cpp\temp> |
```

图 1.11: 链接成功



Chapter 2

Move semantics and rvalue references: Modern C++ fundamentals

👤 Šimon Tóth 📅 2022-04-29 💬★★★

Welcome to the Modern C++ fundamentals series, where we take a deep dive into one topic at a time. Today, we cover move semantics, value categories, and rvalue references introduced in C++11.

Unfortunately, this is one of the topics where we have to start with a bit of front-loaded theory. Notably, we need to talk about value categories.

2.1 The value categories

- *lvalues*: expressions that have an identity; that is, it is possible to determine whether two expressions refer to the same underlying entity
- *rvalues*: expressions that can be “moved from”

These two categories combine:

- *lvalues*: have an identity and can’t be moved from
- *xvalues*: have an identity and can be moved from
- *prvalues*: don’t have an identity and can be moved from
- unused category of expressions that don’t have an identity and can’t be “moved from”

2.1.1 lvalue vs prvalue

To demonstrate, we will contrast *lvalues* and *prvalues*:

```
1 int x = 30, y = 30;
2 assert(&x != &y);
```

The variable names *x* and *y* are *lvalues*. Notably, any name of a variable, function, template parameter object or data member is an *lvalue*.

The variables `x` and `y` are distinct entities, and indeed we can verify this fact using an `assert` (line 2). The integer constants are *prvalues*, both of the constants have the same meaning, but it does not make sense to discuss whether they are one or multiple entities.

```
1 int x = 30;
2 assert(&[](int&v) -> int& { return v; }(x) == &x);
```

Importantly, it doesn't matter how complex the expression is. As long as it maintains identity, the expression is an *lvalue*. So here, the inline *lambda call* is an *lvalue* expression. We can also verify that the call result and the name `x` refer to the same entity with an `assert`. However, the lambda itself does not have an identity; therefore, it is a *prvalue*.

Another typical example of *lvalues* and *prvalues* are function-call and operator expressions that result in a reference (for *lvalues*) or a non-reference (for *prvalues*).

2.1.2 xvalue

So, the one remaining category to discuss is *xvalues*, which are glvalues designated as expiring (*xvalues* = expiring values).

```
1 void consume_name(std::string name);
2
3 int main() {
4     std::string name = "Simon Toth";
5     consume_name(std::move(name));
6
7     name = "John Doe";
8     std::cout << name << "\n";
9 }
```

Here we manually designate `name` as expiring by using the `std::move` cast (line 5). The consequence is that the only valid operation (in general) on the same entity is to overwrite its state (line 7).

As the specific behaviour can be controlled by overloading move construction and move assignment operators, some classes offer additional guarantees:

```
1 void consume_element(std::unique_ptr<int> el);
2
3 int main() {
4     std::unique_ptr<int> el = std::make_unique<int>(30);
5     consume_element(std::move(el));
6     assert(el == nullptr);
7 }
```

An expired `unique_ptr` is guaranteed to be `nullptr`.

2.1.3 Can be “moved from”

Finally, let's talk more about *rvalues* and what it means to “*move from*” .

Before C++11, when we wanted to manifest a new value from an existing one, our only option was copy-construction. However, consider that when we have an *xvalue*, the content of the source value is destined to expire.

In many cases, making a copy will be inefficient, as we can cannibalise the content of source value. Similarly, *prvalues* can also be cannibalised as they do not have existence past the current expression.

```

1 void manual_swap(std::string& left, std::string& right) {
2     std::string tmp(std::move(left));
3     left = std::move(right);
4     right = std::move(tmp);
5 }
6
7 std::string person = "I'm a person";
8 std::string animal = "I'm an animal";
9 manual_swap(person, animal);
10 // person == "I'm an animal", animal == "I'm a person"

```

In this example, we take advantage of move-construction (line 2) and move-assignment (lines 3, 4) to quickly swap two strings (which will generally involve simply reassigning three 64bit values and no memory allocation).

2.1.4 When to move-cast

Finally, let's discuss when you should be using the `std::move` cast in your code. Fortunately, this one is simple:

Use `std::move` cast when passing an lvalue to a function call (or operator expression), and the state of the underlying entity is no longer needed.

```

1 void some_function(std::string name);
2
3 std::string my_func() {
4     std::string name = "John Doe";
5     some_function(std::move(name));
6     // OK, we no longer need the state
7
8     std::string label = "100% Orange Juice";
9     name = std::move(label);
10    // OK, syntax sugar for name.operator=(std::move(label));
11
12    label = std::move(std::string("Hello World!"));
13    // BAD, std::string("Hello World!") is a prvalue
14
15    return name; // OK, no cast here
16 }

```

Following this rule religiously can potentially pessimise performance when interacting with legacy (or poorly designed) interfaces. However, it's a good baseline. Using the `std::move` cast in other contexts, particularly on *prvalues* or in the return expression, will prevent compiler optimisations and should be avoided.

2.1.5 Values types summary

Main takeaways to remember:

- **lvalue expressions have an identity**

notably, name expressions are lvalues, and any compound expressions that result in a reference to a named entity are also lvalues

- **prvalues do not have an identity**

all literals are prvalues (except for string literals which are lvalues), so are expressions that manifest temporary values

- **xvalues are the result of marking an lvalue expression as expiring**

• **use move cast** on *lvalue* expressions when the content of the underlying variable is no longer needed

2.2 Rvalue references

To write code that can take advantage of move semantics, we need to discuss the other side of the coin: *rvalue* references.

First, let's have a look at how calls get resolved when using the pre-C++11 approach of having reference and const-reference overloads:

```

1 void accepts_int(int& v) {
2     std::cout << "Calling by reference: " << v << "\n";
3 }
4
5 void accepts_int(const int& v) {
6     std::cout << "Calling by const-reference: " << v << "\n";
7 }
8
9 accepts_int(10); // Call by const-reference
10
11 int x = 15;
12 accepts_int(x); // Call by reference
13 accepts_int(std::move(x)); // Call by const-reference
14
15 const int y = 5;
16 accepts_int(y); // Call by const-reference

```

As you can see, *prvalues* bind to const reference (line 9), modifiable *lvalues* to reference (line 12), *xvalues* to const reference (line 13) and non-modifiable *lvalues* to const reference (line 16).

If we add a third overload that accepts an rvalue reference, the situation will change:

```

1 void accepts_int(int& v) {
2     std::cout << "Calling by reference: " << v << "\n";
3 }
4
5 void accepts_int(const int& v) {
6     std::cout << "Calling by const-reference: " << v << "\n";
7 }
8
9 void accepts_int(int&& v) {
10    std::cout << "Calling by rvalue reference: " << v << "\n";
11 }
12
13 accepts_int(10); // Call by rvalue reference
14
15 int x = 15;
16 accepts_int(x); // Call by reference
17 accepts_int(std::move(x)); // Call by rvalue reference
18
19 const int y = 5;
20 accepts_int(y); // Call by const-reference

```

The rules are:

- **rvalues (prvalues and xvalues)** will bind to either const-references or rvalue references but prefer rvalue references
- **modifiable lvalues** will bind to either const-references or references but prefer references
- **non-modifiable lvalues** will bind to const-references only

The following example will probably confuse you:

```

1 void accepts_int(int& v) {
2     std::cout << "Calling by reference: " << v << "\n";
3 }
4
5 void accepts_int(int&& v) {
6     std::cout << "Calling by rvalue reference: " << v << "\n";
7 }
8
9 accepts_int(10); // Calling by rvalue reference
10
11 int&& x = 10; // OK, prvalues bind to rvalue references
12 accepts_int(x); // Calling by reference

```

When we call `accepts_int` with `x`, it resolves to a call by reference, despite `x` being of a type *rvalue* reference to `int`. To understand why we need to go back to the first section of this article. Remember that

any expression with an identity and any name expression is an *lvalue*. Therefore, the x here is an *lvalue*, and it will bind to a reference.

When we write `int&& x = 10`, we take a *prvalue* (the constant 10), giving it a name and a lifetime. Because of this, to the function there is no difference between `int x = 10;` and `int&& x = 10;`. Notably, both are mutable integer variables whose lifetimes exceed the function call.

2.3 Taking advantage of move semantics

So far, we have been meddling with synthetic examples, but now it's time to cover the typical use case of move semantics, implementing move semantics for your classes.

Suppose your class is implementing custom resource management. In that case, you can likely take advantage of move semantics by implementing a move constructor and move assignment on top of the typical copy constructor, copy assignment and destructor.

Here is an example of a simple stack implementation with move semantics:

```
1 class Stack {
2 public:
3     Stack() : data_(nullptr), size_(0), capacity_(0) {}
4
5     Stack(const Stack& other) : data_(new int[other.capacity_]),
6                                     size_(other.size_), capacity_(other.capacity_) {
7         std::copy(other.data_, other.data_ + other.size_, data_);
8     }
9
10    ~Stack() { delete[] data_; }
11
12    Stack& operator=(const Stack& other) {
13        if (this == &other)
14            return *this;
15
16        int* buff = data_;
17        data_ = new int[other.capacity_];
18        std::copy(other.data_, other.data_ + other.size_, data_);
19        size_ = other.size_;
20        capacity_ = other.capacity_;
21        delete[] buff;
22        return *this;
23    }
24
25    Stack(Stack&& other) : data_(std::exchange(other.data_, nullptr)),
26                             size_(std::exchange(other.size_, 0)),
27                             capacity_(std::exchange(other.capacity_, 0)) {}
28
```

```
29     Stack& operator=(Stack&& other) {
30         if (this == &other)
31             return *this;
32
33         delete[] data_;
34         data_ = std::exchange(other.data_, nullptr);
35         size_ = std::exchange(other.size_, 0);
36         capacity_ = std::exchange(other.capacity_, 0);
37         return *this;
38     }
39
40     void push(int value) {
41         if (size_ == capacity_) {
42             size_t new_cap = std::max(capacity_*2, UINTMAX_C(64));
43             int* buff = new int[new_cap];
44             std::copy(data_, data_ + size_, buff);
45             delete[] data_;
46             data_ = buff;
47             capacity_ = new_cap;
48         }
49
50         data_[size_] = value;
51         ++size_;
52     }
53
54     int pop() {
55         if (empty())
56             throw std::runtime_error("Can't pop empty stack.");
57         --size_;
58         return data_[size_];
59     }
60
61     int peek() {
62         if (empty())
63             throw std::runtime_error("Can't peek into empty stack.");
64         return data_[size_-1];
65     }
66
67     bool empty() {
68         return size_ == 0;
69     }
70
```

```

71 private:
72     int *data_;
73     size_t size_;
74     size_t capacity_;
75 };

```

We are taking advantage of C++14 `std::exchange`, which shortens the two-step process of `x = o`; `other.x = value`; into a single statement. When you contrast the copy constructor (line 5) vs move constructor (line 25) and copy assignment (line 12) vs move assignment (line 29), you can see the difference between making a copy and cannibalising the content of the other instance.

If your class is not implementing custom resource management, you might be able to stick with the rule of zero:

```

1 struct MyStruct {
2     std::string label;
3     std::vector<int> data;
4 };
5
6 class  MyClass {
7 public:
8     MyClass() : label_("default"), data_{1,2,3,4,5} {}
9 private:
10    std::string label_;
11    std::vector<int> data_;
12 };
13
14 MyStruct x{"default", {1, 2, 3, 4, 5}};
15 MyClass y;

```

As long as you do not declare any custom copy or move constructors, copy or move assignment or destructor, all of these will be provided by the compiler. The caveat here is that the default implementation will make a simple piecewise copy/move, which is only suitable for types that do not implement manual resource management.

Move semantics also unlock the potential to implement move-only types. Move only types are desirable for unique resource handles, e.g. `unique_ptr`.

```

1 struct MoveOnly {
2     MoveOnly() = default;
3     MoveOnly(MoveOnly&&) = default;
4     MoveOnly& operator= (MoveOnly&&) = default;
5 };
6
7 MoveOnly a;
8 MoveOnly b;
9 // b = a; Would not compile

```

```

10 b = std::move(a); // OK, xvalue
11 a = MoveOnly{}; // OK, prvalue
12 // MoveOnly c(b); Would not compile
13 MoveOnly c(std::move(b)); // OK, xvalue

```

Declaring move constructor or move assignment (even as default) disables the default copy constructor and copy assignment. Declaring the move constructor also removes the defaulted default constructor (hence, we re-default it on line 2).

2.4 Type of this

Finally, before C++11, we could overload methods based on whether the instance was constant or mutable. In C++11, we can further overload on whether the instance is an *rvalue*:

```

1 struct Demo {
2     void whoami() & { std::cout << "I'm a modifiable lvalue.\n"; }
3     void whoami() const& { std::cout << "I'm a non-modifiable lvalue.\n"; }
4     void whoami() && { std::cout << "I'm an r-value.\n"; }
5 };
6
7 Demo i;
8 i.whoami(); // modifiable
9
10 const Demo j;
11 j.whoami(); // non-modifiable
12
13 Demo{}.whoami(); // r-value

```

Chapter 3

To Save C, We Must Save ABI

👤 Bugurii 📅 2022-03-13 | 🌐 ★★★★

After that first Firebrand of an article on Application Binary Interface (ABI) Stability, I’m not sure anyone expected this to be the title of the next one, huh? It seems especially bad, given this title is in direct contradiction to a wildly popular C++ Weekly Jason Turner¹ did on the exact same subject.

Not only is Jason 110% thoroughly correct in his take, I deeply and fervently agree with him. My last article on the subject of ABI - spookily titled “Binary Banshees and Digital Demons”² - also displayed how implementers not only back-change the standard library to fit the standard (and not the other way around) when they can get away with it, but also that occasionally threaten the existence of newly introduced features using ABI as a cudgel. But, if I’ve got such a violent hatred for ABI Stability and all of its implications,

why would I claim we need to save it?

Should it not be utterly destroyed and routed from this earth? Is it not the anti-human entity that I claimed it was in my last article? Could it be that I was infected by Big Business™ and Big Money□ and now I’m here to shill out for ABI Stability? Perhaps I’ve on-the-low joined a standard library effort and I’m here as a psychological operation to condition everyone to believing that ABI is good. Or maybe I’ve just finally lost my marbles and we can all start ignoring everything I write!

(Un?)Fortunately, none of that has happened. My marbles are all still there, I haven’t been bought out, and the only standard library I’m working on is my own, locked away in a private repository on a git server in some RAID storage somewhere. But, what I have realized steadily is that no matter how much I agitate and evangelize and etc. etc. for a better standard library, and no matter how many bit containers I write that run circles around MSVC STL’s purely because I get to use 64-bit numbers for my bit operations while they’re stuck on 32-bit for Binary Compatibility reasons³, these systems aren’t going to change their tune just for li’l old me. Or Jason Turner. Or anyone else, really, who’s fed up with losing performance and design space to legacy choices when we quite literally were just not smart enough to be making permanent decisions like this. This doesn’t mean we need to give up, however. After all, there’s more than one way to break an ABI.(see figure 3.1)

Silliness aside, it is important to make sure everyone is up to speed on what an “ABI” really is. Let’s look at ABI - this time, from the C side - and what it prevents us from fixing.

¹<https://youtu.be/By7b19YIv8Q>

²<https://thephd.dev/binary-banshees-digital-demons-abi-c-c++-help-me-god-please>

³<https://ztdidk.readthedocs.io/en/latest/benchmarks/bit.html#details>

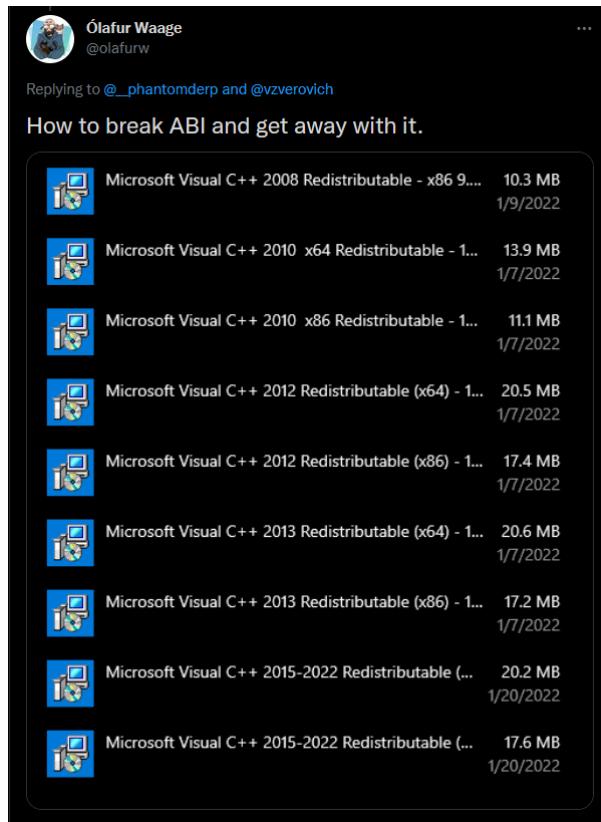


图 3.1: Break ABI

3.1 The Monster - Application Binary Interface

Application Binary Interface, which we will be colloquially referring to as ABI because that’s a whole mouthful to say, is the invisible contract you sign every time you make a structure or write a function in C or C++ code and actually use it to do anything. In particular, it is the assumptions the compiler makes about how exactly the bit-for-bit representation and the usage of the computer’s actual hardware resources when it negotiate things that lie outside of a singular routine. This includes things like:

- the position, order, and layout of members in a **struct/class**;
- the argument types and return type of single function (C++-only: and any relevant overloaded functions);
- the “special members” on a given **class** (C++-only);
- the hierarchy and ordering of virtual functions (C++-only);
- and more.

Because this article focuses on C, we won’t be worrying too much about the C++ portions of ABI. C also has much simpler ways of doing things, so it effectively boils down to two things that matter the most:

- the position, order, and layout of members in a struct; and,
- the argument types and return type of a function.

Of course, because C++ consumes the entire C standard library into itself nearly wholesale with very little modifications, C’s ABI problems become C++’s ABI problems. In fact, because C undergirds way too much software, it is effectively everyone’s problem what C decides to do with itself. How exactly can ABI manifest in C? Well, let’s give a quick example.

3.2 The C ABI

C’s ABI is “simple”, in that there is effectively a one-to-one correspondence between a function you write, and the symbol that gets vomited out into your binary. As an example, if I were to declare a function `do_stuff`, that took a `long long` parameter and returned a `long long` value to use, the code might look like this:

```
#include <limits.h>

extern long long do_stuff(long long value);

int main () {
    long long x = do_stuff(-LLONG_MAX);
    /* wow cool stuff with x ! */
    return 0;
}
```

and the resulting assembly for an x86_64 target would end up looking something like this:

```
main:
movabs rdi, -9223372036854775807
sub    rsp, 8
call   do_stuff
xor    eax, eax
add    rsp, 8
ret
```

This seems about right for a 64-bit number passed in a single register before a function call is made. Now, let’s see what happens if we change the argument from `long long`, which is a 64-bit number in this case, to something like `__int128_t`:

```
#include <limits.h>

extern __int128_t do_stuff(__int128_t value);

int main () {
    __int128_t x = do_stuff(-LLONG_MAX);
    return 0;
}
```

Just a type change! Shouldn’t change the assembly too much, right?

```

main:
sub    rsp, 8
mov    rsi, -1
movabs rdi, -9223372036854775807
call   do_stuff
xor    eax, eax
add    rsp, 8
ret

```

…Ah, there were a few changes. Most notably, we’re not only touching the `rdi` register, we’re messing with `rsi` too. This shows us, already, that without even seeing the inside of the definition of `do_stuff` and how it works, the compiler has forged a **contract** between itself and the people who write the definition of `do_stuff`. For the `long long` version, they expect only 1 register to be used - and it HAS to be `rdi` - on `x86_64` (64-bit) computers. For the `_int128_t` version, they expect 2 registers to be used - `rsi` AND `rdi` - to be used to contain all 128 bits. It sets this up knowing that whoever is providing the definition of `do_stuff` is going to use the **exact same convention**, down to the registers in your CPU. This is not a source code-level contract: it is one forged by the compiler, on your behalf, with other compilers and other machines.

This is the Application Binary Interface.

We note that the problem we highlight is very specific to C and most C-like ABIs. As an example, here is the same `main` with the `_int128_t`-based `do_stuff`’s assembly in C++:

```

main:
push   rax
movabs rdi, -9223372036854775807
mov    rsi, -1
call   _Z8do_stuffn
xor    eax, eax
pop    rcx
ret

```

This `_Z8do_stuffn` is a way of describing that there’s a `do_stuff` function that takes an `_int128_t` argument. Because the argument type gets beaten up into some weird letters and injected into the final function name, the C++ linker can’t be confused about which symbol it likes, compared to the C one. This is called **name mangling**. This post won’t be calling for C to embrace name mangling - no implementation will do that (except for Clang and its `[[overloadable]]` attribute⁴) - which does make what we’re describing substantially easier to go over.

Still, how precarious can C’s direct/non-mangled symbols be, really? Right now, we see that the `call` in the assembly for the C-compiled code only has one piece of information: the name of the function. It just calls `do_stuff`. As long as it can find a symbol in the code named `do_stuff`, it’s gonna call `do_stuff`. So, well, let’s implement `do_stuff`!

⁴<https://clang.llvm.org/docs/AttributeReference.html#overloadable>

3.3 ABI from the Other Side

The first version is the `long long` one, right? We'll make it a simple function: checks if it's negative and returns a specific number (0), otherwise it doesn't do anything. Here's our .c file containing the definition of `do_stuff`:

```
long long do_stuff (long long value) {
    if (value < 0) {
        return 0;
    }
    return value;
}
```

It's kind of like a `clamp`, but only for negative numbers. Either way, let's check what this bad boy puts out:

```
do_stuff:
xor    eax, eax
test   rdi, rdi
cmovns rax, rdi
ret
```

Ooh la la, fancy! We even get to see a `cmovns`! But, all in all, this assembly is just testing the value of `rdi`, which is good! It's then handing it back to the other side in `rax`. We don't see `rax` in the code with `main` because the compiler optimized away our store to `x`. Still, the fact that we're using `rax` for the return is also part of the Application Binary Interface (e.g., not just the parameters, but the return type matters). The compilers chose the same interpretation on both the inside of the `do_stuff` function and the outside of the `do_stuff` function. What does it look like for an `_int128_t`? Here's our updated .c file:

```
_int128_t do_stuff (_int128_t value) {
    if (value < 0) {
        return 0;
    }
    return value;
}
```

And the assembly:

```
1 do_stuff:
2     mov    rdx, rsi
3     xor    esi, esi
4     xor    ecx, ecx
5     mov    rax, rdi
6     cmp    rdi, rsi
7     mov    rdi, rdx
8     sbb    rdi, rcx
9     cmovl rax, rsi
```

```

10 cmove    rdx, rcx
11 ret

```

...Oof. That's a LOT of changes. We see **both** `rsi` and `rdi` being used, we're using a `cmp` (compare) on `rsi` and `rdi`, and we're using a Subtract with Borrow (`sbb`) to get the right computation into the `rcx` register. Not only that, but instead of just using the `rax` register for the return (from `cmove`), we're also applying that to the `rdx` register too (with a similar `cmove`). So there's an expectation of 2 registers containing the return value, not just one! So we've clearly got two different expectations for each set of functions. But...well, I mean, come on.

Can it really break?

How bad would it be if I created an application that compiled with the 64-bit version initially, but was somehow mistakenly linked with the 128-bit version through bad linker shenanigans or other trickery?

3.4 Causing Problems (On Purpose)



Let's see what happens when we break ABI. Our function isn't even that complex; the breakage is probably minor at best! So, here's our 2 .c files:

`main.c:`

```

1 #include <limits.h>
2
3 extern long long do_stuff(long long value);
4
5 int main() {
6     long long x = do_stuff(-LLONG_MAX);
7     /* wow cool stuff with x ! */
8     if (x != 0)
9         return 1;

```

```

10     return 0;
11 }

do_stuff.c:

__int128_t do_stuff(__int128_t value) {
    if (value < 0) {
        return 0;
    }
    return value;
}

```

Now, let's build it, with Clang + MSVC using some default debug flags:

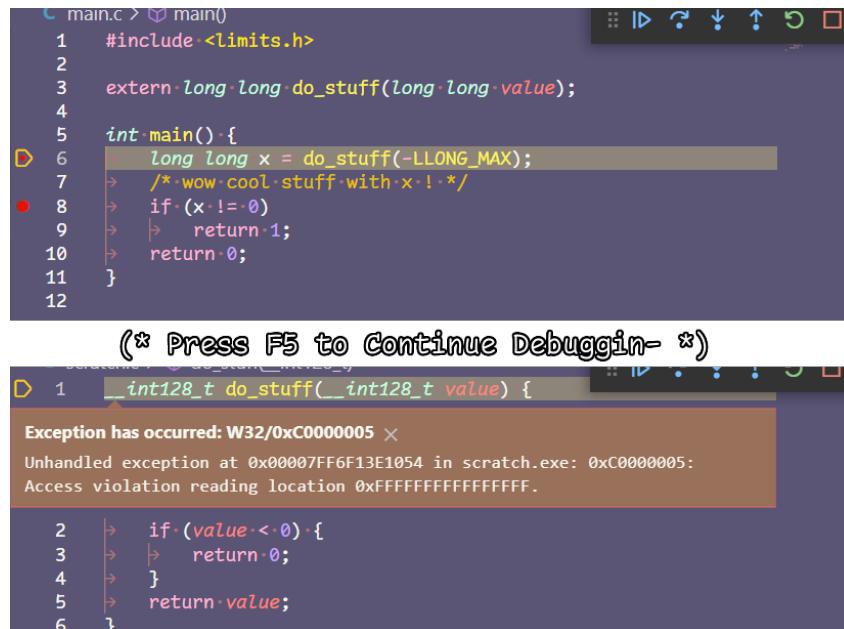
```

[build] [1/2  0% :: 0.009] Re-checking globbed directories...
[build] [2/3  33% :: 0.097] Building C object CMakeFiles/scratch.dir/do_stuff.c.obj
[build] [2/3  66% :: 0.099] Building C object CMakeFiles/scratch.dir/main.c.obj
[build] [3/3 100% :: 0.688] Linking C executable scratch.exe
[build] Build finished with exit code 0

```

[build] Build finished with exit code 0 We'll note that, despite the definition having different types from the declaration, not hiding it behind a DLL, and not doing any other shenanigans to hide the object file that creates the definition from its place of declaration, the linker's attitude to us having completely incompatible declarations and definitions is pretty clear.

But! Even if the linker's fields are barren, assuredly it's not so bad, rig—



Oh. ...

...

Oh.

Okay, so in C we can break ABI just by having the wrong types on a function and not matching it up with a declaration. The linker genuinely doesn't care about types or any of that nonsense. If the symbol `do_stuff` exists, it will bind to the symbol `do_stuff`. Doesn't matter if the function signature is completely

wrong: by the time we get to the linker step —and more importantly, to the actual run-my-executable step — “types” and “safety” are just things for losers. It’s undefined behavior, you messed up, time for you to take a spill and get completely wrecked. Of course, every single person is looking at this and just laughing right now. I mean, come on! This is rookie nonsense, who defines a function in a source file, doesn’t put it in a header, and doesn’t share it so that the **front end** of a compiler can catch it?! That’s just some silliness, right? Well, what if I told you I could put the function in a header and it still broke?

What if I told you this exact problem could happen, even if the header’s code read `extern long long do_stuff(long long value);`, and the implementation file had the right declaration and looked fine too?

3.5 Causing Problems (By Accident)

See, the whole point of ABI breaks is they can happen without the frontend or the linker complaining. It’s not just about headers and source files. We have an new entirely source of problems, and they’re called Libraries. As shown, C does not mangle its identifiers. What you call it in the source code is more or less what you get in the assembly, modulo any implementation-specific shenanigans you get into. This means that when it comes to sharing libraries, everybody has to agree and shake hands on exactly the symbols that are in said library.

This is never more clear than on *nix distributions. Only a handful of people stand between each distribution and its horrible collapse. The only reason many of these systems continue to work is because we take these tiny handful of people and put them under computational constraints that’d make Digital Atlas not only shrug, but yeet the sky and heavens into the void. See, these people - the Packagers, Release Managers, and Maintainers for anyone’s given choice of system configuration —have the enviable job of making sure your dynamic libraries match up with the expectations the entire system has for them. Upgraded dynamic libraries pushed to your favorite places —like the `apt` repositories, the `yum` repositories, or the Pacman locations —need to maintain backwards compatibility. Every single package on the system has to use the agreed upon `libc`, not at the source level,

but at the binary level.

My little sample above? I was lucky: I built on debug mode and got a nice little error popup and something nice. Try doing that with release software on a critical system component. Maybe you get a segmentation fault at the right time. If you’re lucky, you’ll get a core dump that actually gives some hint as to what’s exploded. But most of the time, the schism happens far away from where the real problem is. After all, instead of giving an exception it could just mess with the wrong registers, or destroy the wrong bit of memory, or overwrite the wrong pieces of your stack. It’s unpredictable how it will eventually manifest because it works at a level so deeply ingrained and based on tons of assumptions that are completely invisible to the normal source code developer.

When there are Shared Libraries on the system, there are two sources of truth. The one you compile your application against - the header and its exported interfaces - and the shared library, which actually has the symbols and implementation. If you compile against the wrong headers, you don’t get warned that you have the wrong definition of this or that. It just assumes that things with the same name behave in the expected fashion. A lot of things go straight to hell. So much so that it can delay the adoption of useful, necessary features.

Usually by **ten to eleven** years:

- C99 introduced `_Complex` and Variable Length Arrays. They are now optional, and were made that way in C11. (About 12 years.)
- 10% of the userbase is still using Python 2.x in 2019. Python 3 shipped first around 2008. (About 11 years.)
- C++11 `std::string`: banned copy-on-write first in 2008 (potentially finalized the decision in 2010, I wasn't Committee-ing at that time). Linux distributions using GCC and libstdc++ as their central C++ library finally turned it on in 2018/19. (About 10 years.)

3.6 And That's the Good Ending

Remember, that's C++. You know, the language with the "ambitious" Committee that C and Game Developers like to routinely talk smack about (sometimes for really good reasons, and sometimes for pretty bad reasons). The bad ending you can get if you can't work out a compatibility story is that conservative groups - say, the C Committee - will just blow everything up. For example, when the C Committee first learned that `realloc` on many implementations had diverging behavior, they tried to fix it by releasing Defect Report (DR) 400. Unfortunately, DR 400 still didn't close the implementation-defined behavior loop for `realloc` of size 0. After quite a few more years implementing it, then arguing about it, then trying to talk to implementations, this paper⁵ showed up:

Zero-size Reallocations are Undefined Behavior

Reply-to: Robert C. Seacord (resecord@gmail.com)
 Document No: n2464
 Reference Document: N2433
 Date: 2019-10-25

Summary of Changes

n2464

- Change 7.22.3.5 p3 to indicate that a call to `realloc` where size is zero is undefined behavior

Introduction and Rationale

[DR400](#) was submitted as a result of the divergence of implementations of the `realloc` function as follows:

	Returns	ptr	errno
AIX			
<code>realloc(NULL, 0)</code>	Always <code>NULL</code>		unchanged
<code>realloc(ptr, 0)</code>	Always <code>NULL</code>	<code>freed</code>	unchanged
zOS			
<code>realloc(NULL, 0)</code>	Always <code>NULL</code>		<code>ENOMEM</code>
<code>realloc(ptr, 0)</code>	Always <code>NULL</code>	<code>freed</code>	<code>ENOMEM</code>

You would think adding more undefined behavior to the C Standard would be a bad thing, especially when it was defined before. Unfortunately, implementations diverged and they wanted to stay diverged. Nobody was willing to change their behavior after DR 400. So, of course,

N2464 was voted in unanimously to C23.

In reality, almost everyone on the C Committee is an implementer. Whether it's of a static analysis product, security-critical implementations and guidelines, a (widely available) shipping standard library, an

⁵<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2464.pdf>

embedded compiler, or Linux kernel expertise, there are no greenfield individuals. This is when it became clear that the Standard is not the manifestation of a mandate from on-high, passed down to implementations. Rather, implementations were the ones who dictated everything about what could and could not happen in the C Standard. If any implementation was upset enough about a thing happening, it would not be so.

3.7 But...ABI?

If you read my last article, on ABI, then this is just another spin on exactly what almost happened to Victor Zverovich’s `fntlib` between C++20 and C++23. Threatening intentional non-conformance due to not wanting to / not needing to change behavior is a powerful weapon against any Committee. While the `fntlib` story had a happier ending, this story doesn’t. `realloc` with size of `0` is undefined behavior now, plain and simple, and each implementation will continue to hold the other implementations—and us users—at gunpoint. You don’t get reliable behavior out of your systems, implementations don’t ever have to change (and actively resist such changes), and the standards Committee remains enslaved to implementations.

This means that the Committee generally does not make changes which implementations, even if they have the means to follow, would be deemed too expensive or that they do not want to. That’s the Catch-22. It’s come up a lot in many discussions, especially around `intmax_t` or other parts of the standard. “Well, implementations are stuck at 64-bits because it exists in some function interfaces, so we can’t ever change it.” “Well, we can’t change the return value for these bool-like functions because that changes the registers used.” “So, Annex K is unimplementable on Microsoft because we swapped what the `void*` parameters in the callback mean, and if Microsoft tries to change it to conform to the Standard, we are absolutely ruined.” And so on, and so forth.

But, there is a way out on a few platforms. In fact, this article isn’t about just how bad ABI is, but how to fix at least one part of it, permanently.

3.8 An Old Solution

Any robust C library that is made to work as a system distribution, is already deploying this technique. In fact, if you’re using glibc, musl libc, and quite a few more standard distributions, they can already be made to increase their `intmax_t` to a higher number without actually disturbing existing applications. The secret comes from an old technique that’s been in use with NetBSD for over 25 years⁶: assembly labels.

The link there explains it fairly succinctly, but this allows an implementation to, effectively, rename the symbol that ends up in the binary, without changing your top level code. That is, given this code:

```

1 extern int f(void) __asm__("meow");
2
3 int f (void) {
4     return 1;
5 }
6
7 int main () {
```

⁶https://wiki.netbsd.org/symbol_versions/

```

8     return f();
9 }
```

You may end up with assembly that looks like this:

```

meow:
    mov    eax, 1
    ret
main:
    jmp    meow
```

Notice that the symbol name `f` appears nowhere, despite being the name of the function itself and what we call inside of `main`. What this gives us is the tried-and-true Donald Knuth style of solving problems in computer science: it adds a layer of indirection between what you’re writing, and what actually gets compiled. As shown from NetBSD’s symbol versioning tricks, this is not news: any slightly large operating system has been dealing with ABI stability challenges since their inception. In fact, there are tons of different ways implementations use and spell this:

- Microsoft Visual C: `#pragma comment(linker, "/export:NormalName=_BinarySymbolName")`
- Oracle C: `#pragma redefine_extname NormalName _BinarySymbolName`
- GCC, Arm Keil, and similar C implementations: `Ret NormalName (Arg, Args...) __attribute__((alias("_BinarySymbolName")))`

All of them have slightly different requirements and semantics, but boil down to the same goal. It replaces the `NormalName` at compilation (translation) time with `_BinarySymbolName`, optionally performing some amount of type/entity checking during compilation to prevent connecting to things that do not exist to the compiler’s view (`alias` works this way in particular, while the others will happily do not checking and link to oblivion). These annotations make it possible to “redirect” a given declaration from its original name to another name. It’s used in many standard library distributions, including musl libc. For example, using this `weak_alias` macro and the `__typeof` functionality, musl redeclares several different kinds of names and links them to specifically-versioned symbols within its own binary⁷ to satisfy glibc compatibility:

```

1 #include <stdio.h>
2 #include <stdarg.h>
3
4 int fscanf(FILE *restrict f, const char *restrict fmt, ...)
5 {
6     int ret;
7     va_list ap;
8     va_start(ap, fmt);
9     ret = vfscanf(f, fmt, ap);
10    va_end(ap);
11    return ret;
```

⁷<https://git.musl-libc.org/cgit/musl/tree/src/stdio/fscanf.c>

```

12 }
13
14 weak_alias(fscanf, __isoc99_fscanf);

```

Here, they are using it for compatibility purposes - presenting exactly this symbol name in their binary for the purposes of ABI compatibility with glibc - which begs the question…

why not use it to solve the ABI problem?

If the problem we have in our binaries is that C code built an eon ago expects a very specific symbol name mapped to a particular in-language name, what if we provided a layer of indirection between the symbol name and the in-C-language name? What if we had a Standard-blessed way to provide that layer of indirection? Well, I’m happy to say we don’t have to get academic or theoretical about the subject because I put my hands to the keyboard and figured it out. That’s right,

I developed and tested a solution that works on all 3 major operating system distributions.

3.9 Transparent Aliases

The paper document that describes the work done here is N2901⁸. It contains much of the same statement of the problem that is found in this post, but talks about the development of a solution. In short, what we develop here is a way to provide a symbol that does officially exist as far as the final binary is concerned, much like how the `asm("new_name")` and `__attribute__((alias("old_name")))` are. Notably, the design has these goals:

- it must cost nothing to use;
- it must cost nothing if there is no use of the feature;
- and, it must not introduce a new function or symbol.

Let’s dive in to how we build and specify something like this.

3.10 Zero-Cost (Seriously, We Mean It™ This Time)

Now, if you’ve been reading any of my posts you know that the C Standard loves this thing called “quality of implementation”, or QoI. See, there’s a rule called the “as-if” rule that, so long as the observable behavior of a program (“observable” insofar as the standard provides assurances) is identical, an implementation can commit whatever actions it wants. The idea behind this is that a wide variety of implementations and implementation strategies can be used to get certain work done.

The reality is that known-terrible implementations and poor implementation choices get to live in perpetuity.

Is a compiler allocating something on the heap when the size is predictable and it could be on the stack instead? QoI. Does your nested function implementation mark your stack as executable code rather than dynamic allocation to save space, opening you up to some nasty security vulnerabilities when some buffer overflows get into the mix? QoI. Does what is VERY CLEARLY a set of integer operations meant to be

⁸<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2901.htm>

a rotate left not get optimized into a single `rotl` assembly instruction, despite all the ways you attempt to cajole the code to make the compiler do that?

Quality of Implementation.

Suffice to say, there are a lot of things developers want out of their compilers that they sometimes do and don't get, and the standard leaves plenty of room for those kind of shenanigans. In both developing and standardizing this solution, there must be no room to allow for an implementation to do the wrong thing. Implementation divergence is already a plague amongst people writing cross-platform code, and allowing for a (too) wide a variety of potentially poor implementations helps nobody.

Thusly, when writing the specification for this, we tried to stick as close to “`typedefs`, but for functions” as possible. That is, `typedef` already has all the qualities we want for this feature:

- it costs nothing to use (“it’s just an alias for another type”);
- it costs nothing if there is no use (“just use the type directly if you’re sure”);
- and, it does not introduce a new symbol (e.g. `typedef int int32_t;`, `int32_t` does not show up in C or C++ binaries (modulo special flags and mapping shenanigans for better debugging experiences)).

Thusly, the goal here to achieve all of that for `typedefs`. Because there was such strong existing practice amongst MSVC, Oracle, Clang, GCC, TCC, Arm Keil, and several other platforms, it made it simple to not only write the specification for this, but to implement it in Clang. Here’s an example, using the proposed syntax for N2901:

```
// https://godbolt.org/z/dK5hq5cz
int f(int x) { return x; }

// Proposed: Transparent Aliases
_Alias g = f;

int main () {
    return g(1);
}
```

You’ll see from the generated compiler assembly, that there is no mention of “g” here, with optimizations on OR off! Here is the `-O0` (no optimizations) assembly:

```
1 f: # @f
2     push rbp
3     mov rbp, rsp
4     mov dword ptr [rbp - 4], edi
5     mov eax, dword ptr [rbp - 4]
6     pop rbp
7     ret
8 main: # @main
9     push rbp
10    mov rbp, rsp
```

```

11    sub rsp, 16
12    mov dword ptr [rbp - 4], 0
13    mov edi, 1
14    call f
15    add rsp, 16
16    pop rbp
17    ret

```

And the -O3 (optimizations, including the dangerous ones) assembly:

```

f: # @f
    mov eax, edi
    ret
main: # @main
    mov eax, 1
    ret

```

Now, could a compiler be hostile enough to implement it the worst possible way? Yes, sure, but having an existence proof and writing a specification that strictly conforms to the same kind of transparency allows most non-asshole compiler vendors to do the simple (and correct) thing for the featureset. But why does this work? How does it solve our ABI problem?

3.11 An Indirection Layer

Remember, the core problem with Application Binary Interfaces is that it strongly ties the name of a symbol in your final binary with a given set of semantics. Those semantics - calling convention, register usage, stack space, and even some behaviors - all inform a singular and ultimately unbreakable set of assumptions when code is compiled against a given interface. If you want to change the semantics of a symbol, then, you **must change the name of the symbol**. If you imagine for a second that our **very important ABI symbol** has the name `f` in our code, what we’re aiming to do is to provide a way for new code to access new behaviors and semantics using the same name - `f` - without tying it to behaviors locked into the ABI.

Transparent Aliases are the way to separate the two.

You can provide multiple “internal” (implementation-specific) names in your library for a given function, and then “pick” the right one based on user interaction (e.g., a macro definition). Here’s an example:

```

1 // https://godbolt.org/z/oszMdM6YW
2 extern float __do_work_v0 (float val) { return val; }
3 extern float __get_work_value_v0 (void) { return 1.5f; }
4 extern double __do_work_v1 (double val) { return val; }
5 extern double __get_work_value_v1 (void) { return 2.4; }
6
7 #if VERSION_0
8 typedef float work_t;
9 _Alias do_work = __do_work_v0;
10 _Alias get_work_value = __get_work_value_v0;

```

```

11 #else /* ^^^ VERSION_0 | VERSION_1 or better vvvv */
12 typedef double work_t;
13 _Alias do_work = __do_work_v1;
14 _Alias get_work_value = __get_work_value_v1;
15 #endif
16
17 int main () {
18     work_t v = (work_t)get_work_value();
19     work_t answer = do_work(v);
20     return (int)answer;
21 }

```

And, if we check out the assembly for this:

```

1  __do_work_v0: # @_do_work_v0
2      ret
3  .LCPI1_0:
4      .long 0x3fc00000 # float 1.5
5  __get_work_value_v0: # @_get_work_value_v0
6      movss xmm0, dword ptr [rip + .LCPI1_0] # xmm0 = mem[0],zero,zero,zero
7      ret
8  __do_work_v1: # @_do_work_v1
9      ret
10 .LCPI3_0:
11     .quad 0x4003333333333333 # double 2.3999999999999999
12 __get_work_value_v1: # @_get_work_value_v1
13     movsd xmm0, qword ptr [rip + .LCPI3_0] # xmm0 = mem[0],zero
14     ret
15
16 main: # @main
17     mov eax, 2
18     ret

```

You'll notice that all the functions we wrote implementations for are present. And, more importantly, the symbols `do_work` or `get_work_value` do not appear in the final assembly. A person could compile their top-level application against our code, and depending on the `VERSION_0` macro provide different interfaces (and implementations!) for the code. Note that this means that someone can seamlessly upgrade newly-compiled code to use new semantics, new types, better behaviors, and more without jeopardizing old consumers: the program will always contain the “old” definitions (`..._v0`), until the maintainer decides that it’s time to remove them. (If you’re a C developer, the answer to that question is typically “lol, never, next question nerd”.)

3.12 The True Goal

Most important in this process is that, so long as the end-user is doing things consistent with the described guarantees of the types and functions, their code can be **upgraded for free, with no disturbance in the wider ecosystem at-large**. Without a feature like this in Standard C, it's not so much that implementations are incapable of doing some kind of upgrade. After all, `asm("label")`, `pragma exports`, `--attribute((alias(old_label)))`, and similar all offered this functionality. But the core problem was that because there was no shared, agreed way to solve this problem, implementations that refused to implement any form of this could show up to the C Standards Committee and be well within their rights to give improvements to old interfaces a giant middle finger. This meant that the entire ecosystem suffered indefinitely, and that each vendor would have to - independently of one another - make improvements. If an implementation made a suboptimal choice, and the compiler vendor did not hand them this feature, well. Tough noodles, you and yours get to be stuck in a shitty world from now until the end of eternity.

This also has knock-on effects, of course. `intmax_t` does not have many important library functions in it, but it's unfortunately also tied up to things like the preprocessor. All numeric expressions in the preprocessor are treated and computed under `intmax_t`. Did you want to use 128-bit integers in your code? That's a shame, `intmax_t` is locked to a 64-bit number, and so a strictly-conforming compiler can take a shovel and bash your code's skull in if you try to write a literal value that's bigger than `UINT64_MAX`.

Every time, in the Committee, that we've tried to have the conversation for weaning ourselves off of things like `intmax_t` we've always had problems, particularly due to ABI and the fixed nature of the `typedef` thanks to said ABI. When someone proposes just pinning it down strictly to be `unsigned long long` and coming up with other stuff, people get annoyed. They say "No, I wrote code expecting that my `intmax_t` will grow to keep being the largest integer type for my code, it's not fair I get stuck with a 64-bit number when I was using it properly". The argument spins on itself, and we get nowhere because we cannot form enough consensus as a Committee to move past the various issues.

So, well, can this solve the problem?

Since we have the working feature and a compiler on Godbolt.org that implements the thing (the "thephd.dev" version of Clang), let's try to put this to the test. Can it solve our ABI problems on any of the Big Platforms™? Let's set up a whole test, and attempt to recreate the problems we have on implementations where we pin a specific symbol name to a shared library, and then attempt to fix that binary. The prerequisites for doing this is building the entire Clang compiler with our modifications, but that's the burden of proof every proposal author has these days to satisfy the C Standard Committee, anyways!

3.13 The ABI Test: maxabs

We talked about how `intmax_t` can't be changed because some binary, somewhere, would lose its mind and use the wrong calling convention / return convention if we changed from e.g. `long long` (64-bit integer) to `_int128_t` (128-bit integer). But is there a way that - if the code opted into it or something - we could upgrade the function calls for newer applications while leaving the older applications intact? Let's craft some code that test the idea that Transparent Aliases can help with ABI.

3.14 Making a Shared Library

Our shared library, which we'll just call `my libc` for now, will be very simple. It's going to have a function that computes the absolute value of a number, whose type is going to be of `intmax_t`. First, we need to get some boilerplate out of the way. We'll put this in a typical `<my libc/defs.h>` header file:

```

1 #ifndef MY_LIBC_DEFS_H
2 #define MY_LIBC_DEFS_H
3
4 #if defined(_MSC_VER)
5 #define WEAK_DECL
6 #if defined(MY_LIBC_BUILDING)
7 #define DLL_FUNC __declspec(dllexport)
8 #else
9 #define DLL_FUNC __declspec(dllimport)
10#endif
11#else
12#define WEAK_DECL __attribute__((weak))
13#if defined(_WIN32)
14#if defined(MY_LIBC_BUILDING)
15#define DLL_FUNC __attribute__((dllexport))
16#else
17#define DLL_FUNC __attribute__((dllimport))
18#endif
19#else
20#define DLL_FUNC __attribute__((visibility("default")))
21#endif
22#endif
23
24#if (defined(OLD_CODE) && (OLD_CODE != 0)) || \
25    (!defined(NEW_CODE) || (NEW_CODE == 0))
26#define MY_LIBC_NEW_CODE 0
27#else
28#define MY_LIBC_NEW_CODE 1
29#endif
30
31#endif

```

This is the entire definition file. The most complicated part is working on Windows vs. Everywhere Else™, for the DLL export/import and/or the visibility settings for symbols in GCC/Clang/etc. With that out of the way, let's create the declarations in `<my libc/maxabs.h>` for our code:

```

1 #ifndef MY_LIBC_MAXABS_H
2 #define MY_LIBC_MAXABS_H
3

```

```

4  #include <my_libc/defs.h>
5
6  extern DLL_FUNC int my_libc_magic_number (void);
7
8  #if (MY_LIBC_NEW_CODE == 0)
9      extern DLL_FUNC long long maxabs(long long value) WEAK_DECL;
10     typedef long long intmax_t;
11 #else
12     extern DLL_FUNC __int128_t __libc_maxabs_v1(__int128_t value) WEAK_DECL;
13     typedef __int128_t intmax_t;
14     Alias maxabs = __libc_maxabs_v1; // Alias, for the new code, here!
15 #endif
16
17 #endif

```

We only want one `maxabs` visible depending on the code we have. The first block of code represents the code when we are in the original DLL. It just uses a plain function call, like most libraries would in this day and age. In the new code for the new DLL, we use a new function declaration, coupled with an alias. The concrete function declarations are also marked as a `WEAK_DECL`. This has absolutely no bearing on what we're trying to do here, but we have to keep the code as strongly similar/identical to real-world code as possible, otherwise our examples are bogus. We'll see how this helps us achieve our goal in a brief moment. We pair this header file with:

- one `maxabs.c` file for the original DLL that uses the old code;
- and, both `maxabs.c` and `maxabs.new.c` source files for the new DLL.

Here is the `maxabs.c` file:

```

1  #include <my_libc/defs.h>
2
3  extern DLL_FUNC int my_libc_magic_number (void) {
4  #if (MY_LIBC_NEW_CODE == 0)
5      return 0;
6  #else
7      return 1;
8  #endif
9 }
10
11 // always present
12 extern DLL_FUNC long long maxabs(long long __value) {
13     if (__value < 0) {
14         __value = -__value;
15     }
16     return __value;
17 }

```

When we compile this into the original `my/libc.dll`, we can inspect its symbols. Here's what that looks like on Windows:

```

1 Microsoft (R) COFF/PE Dumper Version 14.31.31104.0
2 Copyright (C) Microsoft Corporation. All rights reserved.
3
4
5 Dump of file abi\old\my_libc.dll
6
7 File Type: DLL
8
9 Section contains the following exports for my_libc.dll
10
11     00000000 characteristics
12         0 time date stamp
13         0.00 version
14         0 ordinal base
15         3 number of functions
16         2 number of names
17
18     ordinal hint RVA           name
19
20         1      0 00001010 maxabs = maxabs
21         2      1 00001000 my_libc_magic_number = my_libc_magic_number

```

The other source file, `maxabs.new.c` is the additional code that provides a new symbol:

```

1 #include <my/libc/defs.h>
2
3 // only for new DLL
4 #if (MY_LIBC_NEW_CODE != 0)
5 extern __int128_t __libc_maxabs_v1(__int128_t __value) {
6     if (__value < 0) {
7         __value = -__value;
8     }
9     return __value;
10 }
11#endif

```

This source file only creates a definition for the new symbol if we've got the proper configuration macro on. And, when we inspect this using `dumpbin.exe` to check for the exported symbols, we see that the `my/libc.dll` in the new directory has the changes we expect:

```

1 Microsoft (R) COFF/PE Dumper Version 14.31.31104.0
2 Copyright (C) Microsoft Corporation. All rights reserved.
3

```

```

4
5 Dump of file abi\new\my_libc.dll
6
7 File Type: DLL
8
9 Section contains the following exports for my_libc.dll
10
11     00000000 characteristics
12         0 time date stamp
13         0.00 version
14         0 ordinal base
15         4 number of functions
16         3 number of names
17
18     ordinal hint RVA           name
19
20     1      0 00001030 __libc_maxabs_v1 = __libc_maxabs_v1
21     2      1 00001010 maxabs = maxabs
22     3      2 00001000 my_libc_magic_number = my_libc_magic_number

```

Note, very specifically, that the old `maxabs` function is still there. This is because we marked its definition in the `maxabs.c` source file as both `extern` and `DLL_FUNC` (to be exported). But, critically, it is not the same as the alias. Remember, when the compiler sees `_Alias maxabs = __libc_maxabs_v1;`, it simply produces a “`typedef`” of the function `__libc_maxabs_v1`. All code that then uses `maxabs`, as it did before, will just have the function call transparently redirected to use the new symbol. This is the most important part of this feature: it is not that it should be a transparent alias to the desired symbol. It is that it must be, so that we have a way to transition old code like the one above to the new code. But, speaking of that transition…now we need to check if this can work in the wild. If you do a drop-in replacement of the old `libc`, do old applications - that cannot/will not be recompiled - still use the old symbols despite having the new DLL and its shiny new symbols present? Let’s make our applications, and find out.

3.15 The “Applications”

We need some source code for the applications. Nothing terribly complicated, just something to use the code (through our shared library) and prove that, if we swap the DLL out from under the application, it continues to work as if we’ve broken nothing. So, here’s an “app” :

```

1 #include <my_libc/maxabs.h>
2
3 #include <stdio.h>
4
5 int main() {
6     intmax_t abi_is_hard = -(intmax_t)sizeof(intmax_t);
7     intmax_t but_not_that_hard = maxabs(abi_is_hard);

```

```

8     printf("%d\n", my_libc_magic_number());
9     return (but_not_that_hard == ((intmax_t)sizeof(intmax_t))) ? 0 : 1;
10 }

```

We use a negated `sizeof(intmax_t)` since, for all the platforms we will test on, we have 2's complement integers. This flips most of the bits in the integers to represent the small negative value. If something terrible happens - for example, some registers are not properly used because we created an ABI break - then we'll see that reflected in the result even if we build with optimizations on and no stack protections (-O3 (GCC, Clang, etc.) or -O2 -O2 (MSVC)). Passing that value to the function and not having the expected positively-sized result is a fairly solid, standards-compliant check.

We also print out the magic number, to get a "true" determination of which shared library we are linked against at runtime (since it uses the same function name with no aliasing, we should see 0 for the old library and 1 for the new library, regardless of whether it's the old application or the new application.)

Using the single `app.c` code above, we create 3 executables:

0. Application that was created with `OLD_CODE` defined, and is linked with the `OLD_CODE`-defined shared library. It represents today's applications, and the "status quo" of things that come packaged with your system today.
1. Application that was created with `NEW_CODE` defined, and is linked to a shared library built with `NEW_CODE` defined. This represents tomorrow's applications, which build "cleanly" in the new world of code.
2. Application that was created with `OLD_CODE` defined, but is linked to a shared library built with `NEW_CODE`. This represents today's applications, linked against tomorrow's shared library (e.g., a partial upgrade from "apt" that does not re-compile the world for a new library).

Of importance is case #2. This is the case we have trouble with today and what allows implementations to come to WG14 Standard Meetings and block any kind of progress regarding `intmax_t` or any other "ABI Breaking" subjects today. So, for Application #0, we compile with `#define OLD_CODE` on for the whole build. For Application #1, we compile with `#define NEW_CODE` on for the whole build.

For Application #2, we don't actually compile anything. We create a new directory and place the old application (from #0) and the new DLL (from #1) in the same folder. This triggers what is known as "app local deployment", and the DLL in the local directory for the application will be picked up by the system's application loader. This also works on OSX and Linux, provided you modify the RPATH linker setting to include the extremely special string `${ORIGIN}`, exactly like that, uninterpreted. (Which is a little harder to do in CMake, unless the special raw string [= `this` will be used literally]=] syntax is used.)

It took a while, but I've effectively mocked this up in a CMake Project to make sure that Transparent Aliases worked⁹. It took a few tries to set it up properly, but after setting up the CMake, the tests, and verifying the integrity of the bits, I checked several operating systems. Here's the output on...

Windows:

```

1 transparent-aliases> abi\old\app_old.lib_old.exe
2 0
3 transparent-aliases> $LASTEXITCODE

```

⁹<https://github.com/ThePhD/transparent-aliases>

```

4  0
5
6 transparent-aliases> abi\new\app_new.lib_new.exe
7 1
8 transparent-aliases> $LASTEXITCODE
9 0
10
11 transparent-aliases> abi\new\app_old.lib_old.exe
12 1
13 transparent-aliases> $LASTEXITCODE
14 0

```

and, on Ubuntu:

```

1 > ./abi/old/app_old.lib_old
2 0
3 > $?
4 0
5
6 > ./abi/new/app_new.lib_new
7 1
8 > $?
9 0
10
11 > ./abi/new/app_old.lib_old
12 1
13 > $?
14 0

```

Perfect. The return code of 0 (determined with \$LASTEXITCODE in Powershell and \$? in Ubuntu's zsh) lets us know that the negative value passed into the function was successfully negated, without accidentally breaking anything (only passing half a value in a register or referring to the wrong stack location entirely). The last executable invoked from the command line corresponds to the case we discussed for #2, where we're in the new/ directory. This directory contains the mylibc.dll/mylibc.so, and as we can see from the printed number invoked after each executable, we get the proper DLL (0 for the old one, 1 for the new one).

Thus, I have successfully synthesized a language feature in C capable of providing backwards binary compatibility with shared libraries/global symbol tables!

…But, unfortunately, as much as I'd like to spend the rest of this post celebrating, it's not all rainbows and sunshine.

3.16 No Free Lunch

Yeah, sometimes some things are too good to be true.

What I've proposed here does not fix all scenarios. Some of them are just the normal dependency management issues. If you build a library on top of something else that uses one of the changed types (such

as `intmax_t` or something else), then you can't really upgrade until your dependents do. This requirement does not exist for folks who typically compile from source, or folks who have things built tailor-made for themselves: usually, your embedded developers and your everything-must-be-a-static-library-I-can-manage types of people. For those of us in large ecosystems who have to write plugins or play nice with other applications and system libraries, we're generally the last to get the benefits. But,

at least we'll finally have the chance to have that discussion with our communities, rather than just being outright denied the opportunity before Day 0.

There's also one other scenario it can't help. Though, I don't think **anyone** can help fix this one, since it's an explicit choice Microsoft has made. Microsoft's ABI requirements are so painfully restrictive that they not only require backwards compatibility (old symbols need to be present and retain the same behavior), but forward compatibility (you can downgrade the library and "strip" new symbols, and **newly built** code must still work with the downgraded shared library). The solution that the Microsoft STL has adopted is on top of having files like `msvcp140.dll`, whenever they need to break something they ship an entirely new DLL instead, even if it contains literally only a single object such as `msvc140p_atomic_wait.dll`, `msvc140p_1.dll`, and `msvc140p_2.dll`. Some of them contain almost no symbols at all, and now that they are shipped nothing can be added or removed to that list of symbols lest you break a new application that has it's DLL swapped out with an older version somewhere. Poor `msvcp140_codecvt_ids.dll` is 20,344 bytes, and for all that 20 kB of space, its sole job is this:

```

1 Microsoft (R) COFF/PE Dumper Version 14.31.31104.0
2 Copyright (C) Microsoft Corporation. All rights reserved.
3
4 File Type: DLL
5
6 Section contains the following exports for MSVCP140_CODECVT_IDS.dll
7
8 00000000 characteristics
9 E13307D2 time date stamp
10 0.00 version
11 1 ordinal base
12 4 number of functions
13 4 number of names
14
15 ordinal hint RVA           name
16
17 1 00003058 ?id@?$codecvt@_SDU_Mbstetat@0@std@0@V0@locale@0@A
18 2 00003040 ?id@?$codecvt@_S_QU_Mbstetat@0@std@0@V0@locale@0@A
19 3 00003050 ?id@?$codecvt@_UDU_Mbstetat@0@std@0@V0@locale@0@A
20 4 00003048 ?id@?$codecvt@_U_QU_Mbstetat@0@std@0@V0@locale@0@A

```

Whenever they need a new symbol —even if it's more codecvt IDs—they can't just slip it into this relatively sparse DLL: it has to go into an entirely different DLL altogether before being locked into stability from now until the heat death of the Windows ecosystem. Transparent Aliases can't save Windows from this kind of design choice because Transparent Aliases are predicated on the idea that you can add new

symbols, exports, and whatever else to the dynamic library without doing anything to the old ones. But, hey: if Microsoft wants to take RedHat's ABI stability and Turn It Up To 11, who am I to argue with the Billions they're raking in on a yearly basis? Suffice to say, if they ever change their mind, at least Transparent Aliases would be capable of solving their current Annex K predicament! That is, they have a different order for the `void*` parameters that are the userdata pointer. Like, as currently exists, Microsoft's `bsearch_s`:

```

1 void* bsearch_s(const void *key, const void *base,
2     size_t number, size_t width,
3     // Microsoft:
4     int (*compare) (void* userdata, const void* key, const void* value),
5     void* userdata
6 );
7
8 void* bsearch_s(const void *key, const void *base,
9     size_t number, size_t width,
10    // Standard C, Annex K:
11    int (*compare) (const void* key, const void* value, void* userdata),
12    void* userdata
13 );

```

It's one of the key reasons Microsoft can't fully conform to Annex K, and why the code isn't portable between the platforms that do have it implemented. With Transparent Aliases, a platform in a similar position to Microsoft can write a new version of this going forward:

```

1 void* bsearch_s_annex_k(const void *key, const void *base,
2     size_t number, size_t width,
3     int (*compare) (const void* key, const void* value, void* userdata),
4     void* userdata
5 );
6
7 void* bsearch_s_msvc(const void *key, const void *base,
8     size_t number, size_t width,
9     int (*compare) (void* userdata, const void* key, const void* value),
10    void* userdata
11 );
12
13 #if defined(_CRT_SECURE_STANDARD_CONFORMING) && (_CRT_SECURE_STANDARD_CONFORMING != 0)
14     _Alias bsearch_s = bsearch_s_annex_k;
15 #else
16     _Alias bsearch_s = bsearch_s_msvc;
17 #endif

```

This would allow MSVC to keep backwards compatibility in old DLLs, while offering standards-conforming functionality in newer ones. Of course, because of the rule that they can't change existing DLL's exports, there's no drop-in replacement. A new DLL has to be written containing these symbols, and code wishing to take advantage of this has to re-compile anyways.

But, at least there may be a way out, if they so choose, in the future if they perhaps relax some of their die-hard ABI requirements. Still, given the demo above and the it-would-work-if-they-did-not-place-these-limitations-on-themselves-or-just-shipped-a-new-DLL nature of things, I would consider this…

3.17 A Great Success

No longer a theoretical idea, this is an existence proof that we can create backwards-compatible shared libraries on multiple different platforms that allow for seamless upgrading. A layer of indirection between the name the C code sees and uses for its functions versus what the actual symbol name effectively creates a small, cross-platform, compile-time-only symbol preservation mechanism. It has no binary size penalty beyond what you, the end user, decide to use for any additional symbols you want to add to old shared libraries. No old symbols have to be messed with, solving the shared library problem. Having an upgrade path finally stops dragging along the technical liability of things chosen from well before I was even born:

Wow, Y2K wasn’t a bug; it was technical debt.

I think I’m gonna throw up.

—Misty, Senior Product Manager, Microsoft & Host of Retro Tech, March 3 2022

Our forebears are either not interested in a world without the mounting, crushing debt or just prefer not to tackle that mess right now (and may get to it Later™, maybe at the Eleventh Hour). Whether out of necessity for the current platforms, or just not wanting to sit down and really do a “recompile the world” deal, they pass this burden on to the rest of us to deal with. It gets worse, too, when you realize that many of them start to check out and retire (or just straight up burn out). This means that we, as the folks now coming to inherit this landscape, have decisions we need to be making. We can continue to deal with their problems, continue to fight with their code and implementations into 2038 and beyond, continue limiting our imagination and growth for both our standard libraries or our regular libraries for the sake of compatibility …Or.

We can actually fix it.

I’m going to hit my 30s soon. I have no desire to still be talking about `time_t` upgrades when I’m in my 40s and 50s, let alone arguing about why `intmax_t` being stuck at 64-bits is NOT fine, and how it is not NOT okay that 64-bits is the biggest natively-exposed C integer type anyone can get out of Standard C. I didn’t put up with a lifetime of suffering to deal with this in a digital world where we already control all the rules. That this is the best we can do in a place of infinite imagination just outright sucks, and having things like the C Standard stuck in 1989 over these decisions is even worse for those of us who would like to take their systems beyond what has already been done. Therefore…

3.18 I will embrace them.

I will Embrace these aging `imaxabs` and `gmtime` and other such symbols.

I will Extend their functionality and allow new implementations and new librarians able and willing to imagine a better world to alias newer programmers to the delightfully improved functionality while the old stuff hobbles along, old symbols left in their current state of untouchable decay. I will put Transparent Aliases in the C Standard and pave a way for the new.

And when the archival is done? When the old programs are properly preserved and the old guard closes their eyes, well taken care of into their last days? I will arm myself. I will make one more trip down into the depths of the Old and the Dark. I will find each and every one of the last symbols, the 32-bit and 64-bit shackles we have had to live with all these years. And to save us —to save our ABI and the to-be-imagined future —I will…



Extinguish them.

Part II

Modern C++

本 Part 主要包含 Modern C++ 的内容，其中，C++23 的内容单独归类到了 **C++23** 那一 Part，C++20 *Coroutines* 的内容归类到了 **Concurrency** 那一 Part，算法相关内容归类到了 **Algorithms** 那一 Part。

这部分主要包含一些其他特性，文章难度主要在三四星，讨论的概念包含函数演进、*constexpr* 演进、非静态数据成员初始化、多态方式、结构化绑定、*Concepts* 等等，利于理清概念发展脉络，明晰一些概念的注意点。

其实其他 Part 中还含有许多 Modern C++ 的内容，只是它们更加适合放到其他相关主题。因此，想要深入地理解其他 Modern C++ 特性，还要阅读其他 Part。

2023 年 5 月 3 日

404

Chapter 4

The Evolution of Functions in Modern C++

👤 Marius Bancila 📅 2022-01-01 💬 ★★★

In programming, a function is a block of code that performs a computational task. (In practice, people write functions that perform many tasks, which is not very good, but it's a topic beyond the purpose of this article). Functions are a fundamental concept of programming languages and C++ makes no exception. In fact, in C++ there is a large variety of functions that has evolved over time. In this article, I will give a brief walkthrough of this evolution starting with C++11. Since there are many things to talk about, I will not get into too many details on these topics but will provide various links for you to follow if you want to learn more.

Let's start briefly with what he had before "modern" times.

4.1 Pre-C++11

Functions were available since the beginning of C++, whose first variant was called C with classes. This is how a function looks:

```
int add(int a, int b)
{
    return a + b;
}
```

This is what we call a **non-member function** or a **free function**, because it does not belong to any class. There are also member functions, that are part of a class/struct. These are also referred as **methods** (like in most other object-oriented programming languages), although this term is not used anywhere in the C++ standard. Here is an example:

```
1 class math
2 {
3     public:
4         int add(int a, int b)
```

```

5     {
6         return a + b;
7     }
8 };

```

There are multiple kinds of functions, including the following:

- overloaded functions

```

int add(int a, int b) {return a + b;}
double add(double a, double b) {return a + b;}

```

- static functions

```

static int add(int a, int b) {return a + b;}

struct math
{
    static int add(int a, int b) {return a + b;}
}

```

- inline functions

```

inline int add(int a, int b) {return a + b;}

struct math
{
    inline int add(int a, int b);
}

```

```
int match::add(int a, int b) {return a + b;}
```

- operators

```

std::string operator+(std::string const & txt, int n)
{
    return txt + std::to_string(n); // channels your JavaScript energy
}

```

- constant member functions

```

class wrapper
{
public:
    wrapper(int a): value_(a) {}
        int get() const {return value_;}
private:
    int value_;
};

```

- virtual member functions

```

struct A
{
    virtual void f() { std::cout << "A:f()\n"; }

};

struct B : public A
{
    virtual void f() { std::cout << "B:f()\n"; }
};

```

- special class functions (default constructor, copy-constructor, copy-assignment operator, and destructor)

```

class wrapper
{
public:
    wrapper() : value_(0) {}
    wrapper(wrapper const & other) {value_ = other.value_; }
    wrapper& operator=(wrapper const & other) {
        if(this != &other) {value_ = other.value_; }
    }
    ~wrapper() {}
private:
    int value_;
};

```

All these are very simple examples but the point here is not to detail all these features that existed before modern C++. One thing that is missing here, though, is templates. Templates are blueprints that define families of functions or classes. The compiler instantiates actual overloads (in the case of function templates) from their use. Here is an example:

```

template <typename T>
T add(T a, T b)
{
    return a + b;
}

```

Now that we've briefly looked at these, let's see what changes modern C++ brought.

4.2 C++11

4.2.1 Variadic function templates

These are function templates with a variable number of arguments.

```

template <typename T>
T add(T a, T b)
{
    return a + b;
}

template <typename T, typename ...Ts> // [1]
T add(T t, Ts ... rest) // [2]
{
    return t + add(rest...); // [3]
}

```

The ellipsis (...) defines a parameter pack. We can have:

- a template parameter pack, such as **typename ... Ts** at line [1]
- a function parameter pack, such as **Ts ... rest** at line [2]
- a pack expansion, such as **add(rest...)** at line [3]

4.2.2 Alternative function syntax

The return type of a function can be placed at the end of the function declaration, after the -> token:

```

auto add(int a, int b) -> int
{
    return a + b;
}

```

In C++11, this is not of much help for non-template functions, but it's important for some function templates. Consider a version of add() that takes arguments of different types:

```

template<typename T, typename U>
??? add(T const & a, U const & b)
{
    return a + b;
}

```

What should the return type be? With the alternative function syntax we can place the return at the end of the expression and specify it with a **decltype** expression:

```

template<typename T, typename U>
auto add(T const & a, U const & b) -> decltype(a + b)
{
    return a + b;
}

```

4.2.3 `constexpr` functions

These are functions that can be evaluated at compile-time. The result of evaluating such a function is a compile-time value that can be used anywhere compile-time values are required. To make a function `constexpr` you need to define it with the `constexpr` keyword, such as in the following example:

```

1 template <typename T>
2 constexpr T add(T a, T b)
3 {
4     return a + b;
5 }
6 int main()
7 {
8     int arr[add(1,2)] = {1,2,3};      // [1]
9     int a, b;
10    std::cin >> a >> b;
11    std::cout << add(a, b) << '\n'; // [2]
12 }
```

Just because a function is declared `constexpr`, doesn't mean it is evaluated at compile-time. In the above example:

- the first call to `add` is evaluated at compile-time (line [1]) because all its arguments are integer literals
- the second call to `add` (at line [2]) is evaluated at runtime because its arguments are only known at runtime

4.2.4 Override and final specifiers for virtual functions

These new specifiers help us better describe virtual functions in derived classes.

The `override` specifier used on a virtual function tells the compiler it is an overridden function of a base class virtual function. If the signature does not match, the compiler triggers an error.

```

1 struct A
2 {
3     virtual void f(int) {}
4     virtual void g() {}
5 };
6 struct B : public A
7 {
8     void f(int) override {} // OK
9     void g(char) override {} // error, g() does not override anything
10}
```

The `final` specifier tells a compiler a virtual function can longer be overridden in a derived class.

```

1 struct A
2 {
```

```

3     virtual void f() {}
4 };
5 struct B : public A
6 {
7     void f() override final {}
8 };
9 struct C : public B
10 {
11     void f() override {}    // error, f cannot be overridden anymore
12 };

```

It should be mentioned that the `final` specifier can also be used on classes, in which case it prevents a class from being further derived.

4.2.5 More special member functions

Move semantics are not easy to describe in one sentence. Basically, it's a language feature that enables the transfer of ownership of a resource from one object to another. Their purpose is improving performance by avoiding copies of resources that are not really necessary. For classes, these bring two new special functions: `move constructor` and `move assignment operator`:

```

1 struct buffer
2 {
3     buffer()           // default constructor
4         :data_(nullptr), size_(0)
5     {}
6
7     explicit buffer(size_t size) // constructor
8         :data_(new char[size]), size_(size)
9     {}
10
11    ~buffer()          // destructor
12    {
13        delete [] data_;
14    }
15
16    buffer(buffer const & other) // copy constructor
17        : data_(new char[other.size_])
18        , size_(other.size_)
19    {
20        std::memcpy(data_, other.data_, size_);
21    }
22
23    buffer& operator=(buffer const & other) // copy assignment operator

```

```
24     {
25         if(this != &other)
26         {
27             delete [] data_;
28             data_ = new char[other.size_];
29             size_ = other.size_;
30             std::memcpy(data_, other.data_, size_);
31         }
32
33         return *this;
34     }
35
36     buffer(buffer&& other)           // move constructor
37         : data_(std::move(other.data_))
38         , size_(other.size_)
39     {
40         other.data_ = nullptr;
41         other.size_ = 0;
42     }
43
44     buffer& operator=(buffer&& other) // move assignment operator
45     {
46         if(this != &other)
47         {
48             delete [] data_;
49             data_ = std::move(other.data_);
50             size_ = other.size_;
51             other.data_ = nullptr;
52             other.size_ = 0;
53         }
54
55         return *this;
56     }
57
58     private:
59     char* data_;
60     size_t size_;
61 };
62
63 int main()
64 {
65     buffer b1;
```

```

66     buffer b2(10);
67     buffer b3 = b2;
68     buffer b4 = std::move(b3);
69 }
```

4.2.6 Default and deleted functions

The special member functions (see above) can be generated by the compiler. However, this does not happen in some circumstances. For instance, if any user-defined constructor exists, a default constructor is not generated, or if a move constructor or move assignment operator is defined, then no copy constructor and copy assignment operator is generated. Rather than implementing these by yourself you can explicitly ask the compiler to generate the default implementation, using the = `default` specifier.

```

struct foo
{
    foo(int) {}      // user-defined constructor
    foo() = default; // compiler generated default constructor
};
```

On the other hand, sometimes we need some functions or some function overloads to not be available. We can prevent a function from being called by defining it with the = `delete` specifier:

```

struct noncopyable
{
    noncopyable() = default;
    noncopyable(noncopyable const &) = delete;
    noncopyable& operator=(noncopyable const &) = delete;
};
```

Any function can be deleted, not just member functions, or special member functions (as shown in the previous example).

```

1 template <typename T>
2 T add(T a, T b)
3 {
4     return a + b;
5 }
6
7 template <>
8 int add<int>(int a, int b) = delete;
9
10 int main()
11 {
12     add(1, 2); // error, this specialization is deleted
13 }
```

4.2.7 Lambdas

Lambdas are not really functions in C++ and the term `lambda function` is incorrect. The right term is `lambda expressions`. Lambdas are syntactic sugar for creating unnamed function objects (which can capture variables in scope). A function object is a class with an overloaded call operator.

```
1 int main()
2 {
3     auto add = [] (int a, int b) { return a + b; };
4     add(1, 2);
5 }
```

The compiler would generate something as follows (conceptually, as the details may vary):

```
1 int main()
2 {
3     class __lambda_1_10
4     {
5         public:
6             inline int operator()(int a, int b) const
7             {
8                 return a + b;
9             }
10    };
11
12    __lambda_1_10 add = __lambda_1_10 {};
13    add.operator()(1, 2);
14 }
```

Lambdas are useful for encapsulating a few lines of code that are then passed to functions such as general purpose algorithms or asynchronous functions.

```
1 int main()
2 {
3     std::vector<int> v {1, 5, 9, 2, 7};
4
5     std::sort(v.begin(), v.end(), [] (int a, int b){return a > b;}); // sorts descending
6
7     for(const auto & e : v)
8         std::cout << e << '\n';
9 }
```

4.3 C++14

4.3.1 Function return type deduction

The alternative function syntax with trailing return type got simplified in C++14 with the compiler being able to deduce the return type from the return expression(s) present in the body of a function. Therefore, functions can be simplified as follows:

```
auto add(int a, int b)
{
    return a + b;
}
```

Again, this is more useful in template code:

```
template <typename T, typename U>
auto add(T a, U b)
{
    return a + b;
}
```

4.3.2 Generic lambdas

A generic lambda is a lambda expression with at least one parameter specified with the `auto` specifier.

```
int main()
{
    using namespace std::string_literals;

    auto add = [] (auto a, auto b) {return a + b;};

    add(1, 2);
    add(1.0, 2.0);
    add("1"s, "2"s);
}
```

This has the effect that the anonymous structure generated by the compiler has a template function call operator. For the above example, it would look, at least conceptually, as follows:

```
1 int main()
2 {
3     using namespace std::string_literals;
4
5     class __lambda_8_16
6     {
7         public:
8             template <typename T0, typename T1>
```

```

9     inline auto operator()(T0 a, T1 b) const
10    {
11        return a + b;
12    }
13
14    template<>
15    inline int operator()(int a, int b) const
16    {
17        return a + b;
18    }
19    template<>
20    inline double operator()(double a, double b) const
21    {
22        return a + b;
23    }
24    template<>
25    inline std::string operator()(std::string a, std::string b) const
26    {
27        return std::operator+(a, b);
28    }
29    };
30
31    __lambda_8_16 add = __lambda_8_16{};
32    add.operator()(1, 2);
33    add.operator()(1.0, 2.0);
34    add.operator()(std::operator""s("1", 1UL), std::operator""s("2", 1UL));
35 }
```

4.4 C++20

4.4.1 Immediate functions

Constexpr functions from C++11 can be evaluated either at compile-time (if all arguments are compile-time values) or runtime. C++20 adds a new categories of functions, called **immediate functions**, that must be evaluated at compile-time. They always produce a compile-time expression and they are always visible only at compile-time. Symbols are not emitted for these functions, you cannot take the address of such functions, and tools such as debuggers will not be able to show them.

These functions are defined using the new **consteval** keyword. Here is an example:

```

1 consteval int add(int const a, int const b)
2 {
3     return a + b;
4 }
5 int main()
```

```

6  {
7      constexpr int s1 = add(1, 2);    // OK, compile-time evaluation
8      int a = 12, b = 66;
9      const int s2 = add(a, b);      // error
10
11     using fptr = int(int, int);
12     fptr* padd = add;           // error
13 }
```

A **consteval** specifier implies **inline**. A function that is **consteval** is a **constexpr** function, and must satisfy the requirements applicable to **constexpr** functions (or **constexpr** constructors).

4.4.2 Abbreviated function templates

If you find template syntax ugly or difficult this feature is for you. It allows you to write function templates without using template syntax. Instead, you use the **auto** specifier to define function parameters. A function with at least one parameter specified with the **auto** specifier is an abbreviated function template:

```

auto add(auto a, auto b)
{
    return a + b;
}
```

The compiler transforms this into a function template:

```

template <typename T, typename U>
auto add(T a, U b)
{
    return a + b;
}
```

These are actually called **unconstrained abbreviated function templates** because there are no constraints on the template arguments. However, you can specify constraints with the help of concepts. Such functions are called **constrained abbreviated function templates**.

```

auto add(std::integral auto a, std::integral auto b)
{
    return a + b;
}
```

This is the same as follows:

```

template <std::integral T, std::integral U>
auto add(T a, U b)
{
    return a + b;
}
```

4.4.3 Lambda templates

The generic lambdas in C++14 have some shortcomings. For instance, consider this lambda:

```
auto add = [](auto a, auto b) {return a + b;};
```

The compiler generates the following function object:

```
struct _lambda_1
{
    template <typename T0, typename T1>
    inline auto operator()(T0 a, T1 b) const
    {
        return a + b;
    }
};
```

But what if the intention is that the two arguments, `a` and `b`, to be of the same type? There is no way to model that in C++14. For this reason, C++20 introduces lambda template, that allows us to define generic lambdas using template syntax:

```
auto add = []<typename T>(T a, T b) {return a + b;};
```

4.4.4 constexpr virtuals

You heard it right: in C++20, virtual functions can be defined as `constexpr`:

```
1 struct magic
2 {
3     constexpr virtual int def() const { return 0; }
4 };
5
6 struct programming_magic : public magic
7 {
8     constexpr int def() const override { return 42; }
9 };
10
11 constexpr int initval(magic const & m)
12 {
13     return m.def() + 1;
14 }
15
16 int main()
17 {
18     constexpr programming_magic pm;
19     int arr[initval(pm)] = {0};
20 }
```

This doesn't seem to have too many use-cases. I don't see where we can use this too much, but it's now possible.

4.4.5 Coroutines

This one is one of the major features of the C++20 standard. A coroutine is a function that has the ability to be suspended and resumed. Unfortunately, C++20 only defines a framework for the execution of coroutines, but does not define any coroutine types satisfying such requirements. That means, we need to either write our own or rely on 3rd party libraries for this. Such a library is the `cppcoro`¹ library.

In C++20, there are three new keywords, for coroutines: `co_await`, `co_return`, and `co_yield`. A function becomes a coroutine if it uses one of these three:

- the `co_await` operator to suspend execution until resumed
- the `co_return` keyword to complete execution and optionally return a value
- the `co_yield` keyword to suspend execution and return a value

Here is an example of a producer-consumer scenario (a coroutine produces new values and another coroutine consumes them as they become available):

```

1 #include <cppcoro/generator.hpp>
2
3 cppcoro::generator<std::string> produce_items()
4 {
5     while (true)
6     {
7         auto v = rand();
8         using namespace std::string_literals;
9         auto i = "item"s + std::to_string(v);
10        print_time();
11        std::cout << "produced " << i << '\n';
12        co_yield i;
13    }
14 }
15
16 #include <cppcoro/task.hpp>
17
18 cppcoro::task<> consume_items(int const n)
19 {
20     int i = 1;
21     for(auto const& s : produce_items())
22     {
23         print_time();
24         std::cout << "consumed " << s << '\n';

```

¹<https://github.com/lewissbaker/cppcoro/>

```
25     if (++i > n) break;
26 }
27 co_return;
28 }
```

That's about it for the time being. If I missed anything important, please let me know.

Chapter 5

Design and evolution of `constexpr` in C++

● Evgeny Shulgin 📅 2022-01-13 🔖 ★★★★☆

`constexpr` is one of the magic keywords in modern C++. You can use it to create code, that is then executed before the compilation process ends. This is the absolute upper limit for software performance.

We published and translated this article with the copyright holder's permission. The author is Evgeny Shulgin, email - izaronplatz@gmail.com. The article was originally published on Habr. We'd also like to invite you to read other theoretical articles that have a hashtag #Knowledge.

`constexpr` gets new features every year. At this time, you can involve almost the entire standard library in compile-time evaluations. Take a look at this code¹: it calculates the number under 1000 that has the largest number of divisors.

```
1 #include <vector>
2 #include <algorithm>
3 #include <iostream>
4
5 // returns the number of divisors of `v`
6 constexpr int find_divisors_count(int v) {
7     int res = 0;
8     for (int i = 1; i <= v; ++i) {
9         if (v % i == 0) {
10             ++res;
11         }
12     }
13     return res;
14 }
15
16 // returns the biggest pair {divisors count of `v`, `v`}
```

¹<https://godbolt.org/z/MYTbbsqvT>

```

17 constexpr std::pair<int, int> find_most_divisible(int n) {
18     std::vector<std::pair<int, int>> vec(n + 1);
19     for (int i = 1; i <= n; ++i) {
20         vec[i] = {find_divisors_count(i), i};
21     }
22     std::sort(vec.begin(), vec.end());
23     return vec.back();
24 }
25
26 int main() {
27     static_assert(find_most_divisible(1000) == std::make_pair(32, 840));
28 }
```

`constexpr` has a long history that starts with the earliest versions of C++. Examining standard proposals and compilers' source code helps understand how, layer by layer, that part of the language was created. Why it looks the way it does. How `constexpr` expressions are evaluated. Which features we expect in the future. And what could have been a part of `constexpr` - but was not approved to become part of the standard.

This article is for those who do not know about `constexpr` yet - and for those who've been using it for a long time.

5.1 C++98 and C++03: Ranks among const variables

In C++, sometimes it's necessary to use integer constants, whose values must be available at compile time. The standard allows you to write constants in the form of simple expressions, as in the code below:

```

1 enum EPlants
2 {
3     APRICOT = 1 << 0,
4     LIME = 1 << 1,
5     PAPAYA = 1 << 2,
6     TOMATO = 1 << 3,
7     PEPPER = 1 << 4,
8     FRUIT = APRICOT | LIME | PAPAYA,
9     VEGETABLE = TOMATO | PEPPER,
10 };
11
12 template<int V> int foo();
13 int foo6 = foo<1+2+3>();
14 int foo110 = foo<(1 < 2) ? 10*11 : VEGETABLE>();
15
16 int v;
17 switch (v)
18 {
19     case 1 + 4 + 7:
```

```

20 case 1 << (5 | sizeof(int)):
21 case (12 & 15) + PEPPER:
22     break;
23 }
```

These expressions are described in the [expr.const] section and are called constant expressions. They can contain only the following:

- Literals² (this includes integers, these are integral types);
- *enum* values;
- An *enum* or integral non-type template parameter (for example, the V value from template <int V>);
- The *sizeof* expression;
- const variables initialized by a constant expression –**this is the interesting point**.

All the points except the last one are obvious – they are known and can be accessed at compile time. The case with variables is more intriguing.

For variables with static storage duration, in most cases, memory is filled with zeros and is changed at runtime. However, it is too late for the variables from the list above –their values need to be evaluated before compilation is finished.

There are two types of static initialization in the C++98/03 standards:

1. *zero-initialization*, when memory is filled with zeros and the value changes at runtime;
2. *initialization with a constant expression*, when an evaluated value is written to the memory at once (if needed).

Note. All other initializations are called dynamic initialization, we do not review them here.

Note. A variable that was zero-initialized, can be initialized again the "normal" way. This will already be dynamic initialization (even if it happens before the main method call).

Let's review this example with both types of variable initialization:

```

1 int foo()
2 {
3     return 13;
4 }
5
6 const int test1 = 1 + 2 + 3 + 4; // initialization with a const. expr.
7 const int test2 = 15 * test1 + 8; // initialization with a const. expr.
8 const int test3 = foo() + 5;      // zero-initialization
9 const int test4 = (1 < 2) ? 10 * test3 : 12345; // zero-initialization
10 const int test5 = (1 > 2) ? 10 * test3 : 12345; // initialization with
11                                         // a const. expr.
```

You can use variables test1, test2, test5 as a template parameter, as an expression to the right of case in switch, etc. You cannot do this with variables test3 and test4.

As you can see from requirements for constant expressions and from the example, there is transitivity. If some part of an expression is not a constant expression, then the entire expression is not a constant expression.

²<https://eel.is/c++draft/lex.literal>

Note that only those expression parts, that are evaluated, matter – which is why test4 and test5 fall into different groups.

If there's nowhere for a constant expression variable to get its address, the compiled program is allowed to skip reserving memory for the variable – so we will force the program to reserve the memory anyway. Let's output variable values and their addresses:

```

1 int main()
2 {
3     std::cout << test1 << std::endl;
4     std::cout << test2 << std::endl;
5     std::cout << test3 << std::endl;
6     std::cout << test4 << std::endl;
7     std::cout << test5 << std::endl;
8
9     std::cout << &test1 << std::endl;
10    std::cout << &test2 << std::endl;
11    std::cout << &test3 << std::endl;
12    std::cout << &test4 << std::endl;
13    std::cout << &test5 << std::endl;
14 }
15
16 izaron@izaron:~/cpp$ clang++ --std=c++98 a.cpp
17 izaron@izaron:~/cpp$ ./a.out
18 10
19 158
20 18
21 180
22 12345
23 0x402004
24 0x402008
25 0x404198
26 0x40419c
27 0x40200c

```

Now let's compile an object file and look at the table of symbols:

```

1 izaron@izaron:~/cpp$ clang++ --std=c++98 a.cpp -c
2 izaron@izaron:~/cpp$ objdump -t -C a.o
3 a.o:      file format elf64-x86-64
4
5 SYMBOL TABLE:
6 0000000000000000 1 df *ABS*  0000000000000000 a.cpp
7 0000000000000080 1 F .text.startup  0000000000000015 _GLOBAL__sub_I_a.cpp
8 0000000000000000 1 O .rodata        0000000000000004 test1

```

```

9  00000000000000000004 1    0 .rodata      00000000000000000004 test2
10 00000000000000000004 1    0 .bss       00000000000000000004 test3
11 00000000000000000008 1    0 .bss       00000000000000000004 test4
12 00000000000000000008 1    0 .rodata      00000000000000000004 test5

```

The compiler –its specific version for a specific architecture –placed a specific program’s zero-initialized variables into the .bss³ section, and the remaining variables into the .rodata section.

Before the launch, the bootloader loads the program in a way that the .rodata section ends up in the read-only segment. The segment is write-protected at the OS level.

Let’s try to use *const_cast* to edit data stored at the variables’ address. The standard is not clear as to when using *const_cast* to write the result can cause undefined behavior. At least, this does not happen when we remove const from an object/a pointer to an object that is not fundamentally constant initially. I.e. it’s important to see a difference between physical constancy and logical constancy.

The UB sanitizer catches UB (the program crashes) if we try to edit the .rodata variable. There is no UB if we write to .bss or automatic variables.

```

1 const int &ref = testX;
2 const_cast<int&>(ref) = 13; // OK for test3, test4;
3                               // SEGV for test1, test2, test5
4 std::cout << ref << std::endl;

```

Thus, some constant variables are ”more constant” than others. As far as we know, at that time, **there was no simple way** to check or monitor that a variable had been *initialized with a const. expr.*

5.2 0-∞: Constant evaluator in compiler

To understand how constant expressions are evaluated during compilation, first you need to understand how the compiler is structured.

Compilers are ideologically similar to each other. I’ll describe how Clang/LLVM evaluates constant expressions. I copied basic information about this compiler from my previous article:⁴

5.2.1 [SPOILER BLOCK BEGINS]

5.2.1.1 Clang and LLVM

Many articles talk about Clang and LLVM. To learn more about their history and general structure, you can read this article⁵ at Habr.

The number of compilation stages depends on who explains the compiler’s design. The compiler’s anatomy is multilevel. At the most abstract level, the compiler looks like a fusion of three programs:

- **Front-end:** converts the source code from C/C++/Ada/Rust/Haskell/... into LLVM IR⁶ – a special intermediate representation. Clang is the front-end for the C language family
- **Middle-end:** LLVM IR is optimized depending on the settings.

³<https://en.wikipedia.org/wiki/.bss>

⁴<https://habr.com/en/post/576052/>

⁵<https://habr.com/en/company/huawei/blog/511854/>

⁶<https://llvm.org/docs/LangRef.html>

- **Back-end:** LLVM IR is converted into machine code for the required platform - x86/Arm/PowerPC/...

For simple languages, one can easily write a compiler whose source code consists of 1000 lines⁷ - and get all the power of LLVM - for this, you need to implement the front-end.

At a less abstract level is Clang's front-end that performs the following actions (not including the pre-processor and other "micro" steps):

- **Lexical analysis:** *converting characters into tokens, for example []() return 13 + 37; are converted to (l_square) (r_square) (l_paren) (r_paren) (l_brace) (return) (numeric_constant:13) (plus) (numeric_constant:37) (semi) (r_brace).*
- **Syntactic analysis:** *creating an AST (Abstract Syntax Tree) - that is, translating tokens from the previous paragraph into the following form: (lambdaexpr (body (returnexpr (plusexpr (number 13) (number 37))))).*
- **Code generation:** *creating LLVM IR for specific AST*

5.2.2 [SPOILER BLOCK ENDS]

So, evaluating constant expressions (and entities that are closely related to them, like template instantiation) takes place strictly in the C++ compiler's (Clang's in our case) front-end. LLVM does not do such things.

Let's tentatively call the micro-service that evaluates constant expressions (from the simplest ones in C++98 to the most complicated ones in C++23) the **constant evaluator**.

If, according to the standard, at some location in the code we expect a constant expression; and the expression that is there meets the requirements for a constant expression -Clang must be able to evaluate it in 100% of cases, right then and there.

Constant expression restrictions have been constantly softened over the years, while Clang's constant evaluator kept getting more advanced -reaching the ability to manage the memory model.

Nine-year-old documentation⁸ describes how to evaluate constants in C++98/03. Since constant expressions were very simple then, they were evaluated with the conventional constant folding⁹, through the abstract syntax tree (AST) analysis. Since, in syntax trees, all arithmetic expressions are already broken apart into sub-trees, evaluating a constant is a simple traversal of a sub-tree.

The constant evaluator's source code is located in lib/AST/ExprConstant.cpp¹⁰ and had reached almost 16 thousand lines by the moment I was writing this article. Over the years, it learned to interpret a lot of things, for example, loops (EvaluateLoopBody)¹¹-all of this based on the syntax tree.

The big difference of constant expressions from code executed in runtime - they are required to not allow undefined behavior. If the constant evaluator stumbles upon UB, compilation fails.

```

1 c.cpp:15:19: error: constexpr variable 'foo' must be initialized by a
2           constant expression
3 constexpr int foo = 13 + 2147483647;
4

```

⁷<https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html>

⁸<https://clang.llvm.org/docs/InternalsManual.html>

⁹https://en.wikipedia.org/wiki/Constant_folding

¹⁰https://clang.llvm.org/doxygen/ExprConstant_8cpp_source.html

¹¹https://clang.llvm.org/doxygen/ExprConstant_8cpp.html

The constant evaluator is used not only for constant expressions, but also to look for potential bugs in the rest of the code. This is a side benefit from this technology. Here's how one can detect overflow in non-constant code (you can get a warning):

```
1 c.cpp:15:18: warning: overflow in expression; result is -2147483636
2           with type 'int' [-Winteger-overflow]
3 int foo = 13 + 2147483647;
```

5.3 2003: No need for macros

Changes to the standard occur through proposals.

5.3.1 [SPOILER BLOCK BEGINS]

5.3.1.1 Where are proposals located and what do they consist of?

All proposals to the standard are located at [open-std.org](http://open-std.org/jtc1/sc22/wg21/docs/papers/).¹² Most of them have detailed descriptions and are easy to read. Usually, proposals contain the following:

- A short review of the area with links to standard sections;
- Current problems;
- The proposed solution to the problems;
- Suggested changes to the standard's text;
- Links to previous precursor proposals and previous revisions of the proposal;
- In advanced proposals –links to their implementation in a compiler's fork. For the proposals that I saw, the authors implemented the proposal in Clang's fork.

One can use the links to precursor proposals to track how each piece of C++ evolved.

Not all proposals from the archive were eventually accepted (although some of them were used as a base for accepted proposals), so it's important to understand that they describe some alternative version of C++ of the time, and not a piece of modern C++.

Anyone can participate in the C++ evolution –Russian-speaking experts can use the stdcpp.ru¹³ website.

5.3.2 [SPOILER BLOCK ENDS]

[N1521] Generalized Constant Expressions¹⁴ was proposed in 2003. It points to a problem that if part of an expression is evaluated using a method call, then the expression is not considered a constant expression. This forces developers –when they need a more or less complex constant expression –to overuse macros:

```
1 #define SQUARE(X) ((X) * (X))
2 inline int square(int x) { return x * x; }
3 // ^^^ the macro and method definition
4 square(9)
5 std::numeric_limits<int>::max()
```

¹²<http://open-std.org/jtc1/sc22/wg21/docs/papers/>

¹³<https://stdcpp.ru/en/about>

¹⁴<http://open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1521.pdf>

```

6 // ^^^ cannot be a part of a constant expression
7 SQUARE(9)
8 INT_MAX
9 // ^^^ theoretically can be a part of a constant expression

```

This is why the proposal suggests introducing a concept of constant-valued methods that would be allowed as part of a constant expression. A method is considered constant-valued if this method is inline, non-recursive, does not return void, and its body consists of a single return expr; expression. After substituting arguments (that also include constant expressions), the developer gets a constant expression.

Note. Looking ahead, the term constant-valued didn't catch on.

```

1 int square(int x) { return x * x; }           // constant-valued
2 long long_max(int x) { return 2147483647; } // constant-valued
3 int abs(int x) { return x < 0 ? -x : x; }   // constant-valued
4 int next(int x) { return ++x; }                // NOT constant-valued

```

Thus, all variables from the previous section (test1-5) would become "fundamentally" constant, with no changes in code.

The proposal believes that it's possible to go even further. For example, this code should also compile:

```

1 struct cayley
2 {
3     const int value;
4     cayley(int a, int b)
5         : value(square(a) + square(b)) {}
6     operator int() const { return value; }
7 };
8
9 std::bitset<cayley(98, -23)> s; // eq. to bitset<10133>

```

The reason for this is, the value variable is "fundamentally constant", because it was initialized in a constructor through a constant expression with two calls of the constant valued method. Consequently, according to the proposal's general logic, the code above can be transformed to something like this (by taking variables and methods outside of the structure):

```

1 // imitating constructor calls: cayley::cayley(98, -23) and operator int()
2 const int cayley_98_m23_value = square(98) + square(-23);
3
4 int cayley_98_m23_operator_int()
5 {
6     return cayley_98_m23_value;
7 }
8
9 // creating a bitset
10 std::bitset<cayley_98_m23_operator_int()> s; // eq. to bitset<10133>

```

Proposals do not usually focus deeply on the details of how compilers can implement these proposals. This proposal says that there should not be any difficulties in implementing it - one just needs to slightly alter constant folding, which exists in most compilers.

Note. However, proposals cannot exist in isolation from compilers – proposals impossible to be implemented in a reasonable time are unlikely to be approved.

As with variables, a developer cannot check whether a method is constant-valued.

5.4 2006-2007: When it all becomes clear

Luckily, in three years, over the next revisions of this proposal ([N2235]¹⁵), it became clear that the feature would have brought too much unclarity and this was not good. Then one more item was added to the list of problems - the inability to monitor initialization:

```

1 struct S
2 {
3     static const int size;
4 };
5
6 const int limit = 2 * S::size; // dynamic initialization
7 const int S::size = 256; // constant expression initialization
8 const int z = std::numeric_limits<int>::max(); // dynamic initialization

```

The programmer intended limit to be initialized by a constant expression, but this does not happen, because S::size is defined "too late", after limit. If it were possible to request the required initialization type, the compiler would have produced an error.

Same with methods. Constant-valued methods were renamed to constant-expression methods. The requirements for them remained the same, but now, in order to use these methods in a constant expression, it was necessary to declare them with the `constexpr` keyword. The compilation would fail if the method body is not the correct return expr;.

The compilation would also fail and produce the `constexpr` function never produces a constant expression error if a `constexpr` method cannot be used in a constant expression. This is necessary to help the developer make sure that a method can be potentially used in a constant expression.

The proposal suggests to tag some methods from the standard library (for example, from `std::numeric_limits`) as `constexpr`, if they meet the requirements for `constexpr` methods.

Variables or class members can also be declared as `constexpr` - then the compilation will fail if a variable is not initialized through a constant expression.

At that time, it was decided to keep the new word's compatibility with variables, implicitly initialized through a constant expression, but without the `constexpr` word. Which means the code below worked (looking ahead, this code with `-std=c++11` does not compile – and it is possible that this code never started to work at all):

```

1 const double mass = 9.8;
2 constexpr double energy = mass * square(56.6); // OK, although mass

```

¹⁵<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2235.pdf>

```

3                                     // was not defined
4                                     // with constexpr
5 extern const int side;
6 constexpr int area = square(side); // error: square(side) is not
7                                     // a constant expression

```

Constant-expression constructors for user-defined types were also legalized. This constructor must have an empty body and initialize its members with `constexpr` expressions if a developer creates a `constexpr` object of this class.

The implicitly-defined constructor is marked as `constexpr` whenever possible. Destructors for `constexpr` objects must be trivial, since non-trivial ones usually change something in the context of a running program that does not exist as such in `constexpr` evaluations.

Example of a class with `constexpr` members, from the proposal:

```

1 struct complex
2 {
3     constexpr complex(double r, double i) : re(r), im(i) { }
4
5     constexpr double real() { return re; }
6     constexpr double imag() { return im; }
7
8 private:
9     double re;
10    double im;
11 };
12
13 constexpr complex I(0, 1); // OK -- literal complex

```

The proposal called objects like the `I` object user-defined literals. A "literal" is something like a basic entity in C++. "Simple" literals (numbers, characters, etc) are passed as they are into assembler commands. String literals are stored in a section similar to `.rodata`. Similarly, user-defined literals also have their own place somewhere there.

Now, aside from numbers and enumerations, `constexpr` variables could be represented by literal types introduced in this proposal (so far without reference types). A literal type¹⁶ is a type that can be passed to a `constexpr` function, and/or modified and/or returned from it. These types are fairly simple. Compilers can easily support them in the constant evaluator.

The `constexpr` keyword became a specifier that compilers require – similarly to `override` in classes. After the proposal was discussed, it was decided to avoid creating a new storage class¹⁷ (although that would have made sense) and a new type qualifier¹⁸. Using it with function arguments was not allowed so as not to overcomplicate the rules for overload resolution.

¹⁶https://en.cppreference.com/w/cpp/named_req/LiteralType

¹⁷<https://pvs-studio.com/en/blog/posts/cpp/0909/>

¹⁸<https://en.cppreference.com/w/cpp/language/cv>

5.5 2007: First constexpr for data structures

That year, the [N2349] Constant Expressions in the Standard Library proposal¹⁹ was submitted. It tagged as constexpr some functions and constants, as well as some container functions, for example:

```

1 template<size_t N>
2 class bitset
3 {
4     // ...
5     constexpr bitset();
6     constexpr bitset(unsigned long);
7     // ...
8     constexpr size_t size();
9     // ...
10    constexpr bool operator[](size_t) const;
11 };

```

Constructors initialize class members through a constant expression, other methods contain return expr; in their body. This return expression meets the current requirements.

Over half of the proposals about constexpr talk about tagging some functions from the standard library as constexpr. There are always more proposals like this after each new step of the constexpr evolution. And almost always they are not very interesting.

5.6 2008: Recursive constexpr methods

constexpr methods were not initially intended to be made recursive, mainly because there were no convincing arguments in favor of recursion. Then the restriction was lifted, which was noted in [N2826] Issues with Constexpr.²⁰

```

1 constexpr unsigned int factorial( unsigned int n )
2 {
3     return n==0 ? 1 : n * factorial( n-1 );
4 }

```

Compilers have a certain limit of nested calls. Clang, for example, can process a maximum of 512 nested calls. If this number is exceeded, the compiler won't evaluate the expression.

Similar limits exist for template instantiation (for example, if we used templates instead of constexpr to do compiletime evaluations).

5.7 2010: "const T&" as arguments in constexpr methods

At this time, many functions cannot be tagged as constexpr because of references to constants in the arguments. Parameters are passed by value –i.e. are copied –to all constexpr methods.

¹⁹<https://en.cppreference.com/w/cpp/language/cv>

²⁰<https://open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2826.html>

```

1 template< class T >
2 constexpr const T& max( const T& a, const T& b ); // does not compile
3
4 constexpr pair(); // can use constexpr
5 pair(const T1& x, const T2& y); // cannot use constexpr

```

Proposal [N3039] Constexpr functions with const reference parameters (a summary)²¹ allows constant references in function arguments and as a return value.

This is a dangerous change: before that, the constant evaluator dealt with simple expressions and constexpr variables (a literal-class object – essentially, a set of constexpr variables); but the introduction of references breaks through the "fourth wall", because this concept refers to the memory model that the evaluator does not have.

Overall, working with references or pointers in constant expressions turns a C++ compiler into a C++ interpreter, so various limitations are set.

If the constant evaluator can process a function with a type T argument, processing this function with the const T& is also possible - if the constant evaluator "imagines" that a "temporary object" is created for this argument.

Compilers cannot compile code that requires more or less complicated work or that tries to break something.

```

1 template<typename T> constexpr T self(const T& a) { return *(&a); }
2 template<typename T> constexpr const T* self_ptr(const T& a) { return &a; }
3
4 template<typename T> constexpr const T& self_ref(const T& a)
5 {
6     return *(&a);
7 }
8
9 template<typename T> constexpr const T& near_ref(const T& a)
10 {
11     return *(&a + 1);
12 }
13
14 constexpr auto test1 = self(123); // OK
15 constexpr auto test2 = self_ptr(123); // FAIL, pointer to temporary is not
16 // a constant expression
17 constexpr auto test3 = self_ref(123); // OK
18 constexpr auto test4 = near_ref(123); // FAIL, read of dereferenced
19 // one-past-the-end pointer is not
20 // allowed in a constant expression

```

²¹<http://open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3039.pdf>

5.8 2011: static_assert in constexpr methods

Proposal [N3268] static_assert and list-initialization in constexpr functions²² introduces the ability to write "static" declarations that do not affect how function operate: typedef, using, static_assert. This slightly untightens the nuts for constexpr functions.

5.9 2012: (Almost) any code in constexpr functions

In 2012, there was a big leap forward with the proposal [N3444] Relaxing syntactic constraints on constexpr functions.²³ There are many simple functions that are preferable to be executed at compile-time, for example, the a^n power:

```

1 // Compute a to the power of n
2 int pow(int a, int n)
3 {
4     if (n < 0)
5         throw std::range_error("negative exponent for integer power");
6     if (n == 0)
7         return 1;
8     int sqrt = pow(a, n/2);
9     int result = sqrt * sqrt;
10
11    if (n % 2)
12        return result * a;
13    return result;
14 }
```

However, in order to make its constexpr variant, developers have to go out of their way and write in a functional style (remove local variables and if-statements):

```

1 constexpr int pow_helper(int a, int n, int sqrt)
2 {
3     return sqrt * sqrt * ((n % 2) ? a : 1);
4 }
5
6 // Compute a to the power of n
7 constexpr int pow(int a, int n)
8 {
9     return (n < 0)
10        ? throw std::range_error("negative exponent for integer power")
11        : (n == 0) ? 1 : pow_helper(a, n, pow(a, n/2));
12 }
```

²²<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3268.htm>

²³<http://open-std.org/JTC1/SC22/WG21/docs/papers/2012/n3444.html>

This is why the proposal wants to allow adding any code to `constexpr` functions - with some restrictions:

- It's impossible to use loops (for/while/do/range-based for), because variable changes are not allowed in constant expressions;
- `switch` and `goto` are forbidden so that the constant evaluator does not simulate complex control flows;
- As with the old restrictions, functions should theoretically have a set of arguments that enable you to use these functions in constant expressions. Otherwise, the compiler assumes a function was marked as `constexpr` accidentally, and the compilation will fail with `constexpr` function never produces a constant expression.

Local variables - if they have the literal type - can be declared within these functions. If these variables are initialized with a constructor, it must be a `constexpr` constructor. This way, when processing a `constexpr` function with specific arguments, the constant evaluator can create a "background" `constexpr` variable for each local variable, and then use these "background" variables to evaluate other variables that depend on the variables that have just been created.

Note. There can't be too many of such variables because of a strict limitation on the depth of the nested calls.

You can declare static variables in methods. These variables may have a non-literal type (in order to, for example, return references to them from a method; the references are, however, of the literal type). However, these variables should not have the dynamic realization (i.e. at least one initialization should be a zero initialization). The sentence gives an example where this feature could be useful (getting a link to a necessary object at compile-time):

```

1 constexpr mutex &get_mutex(bool which)
2 {
3     static mutex m1, m2; // non-const, non-literal, ok
4     if (which)
5         return m1;
6     else
7         return m2;
8 }
```

Declaring types (class, enum, etc.) and returning void was also allowed.

5.10 2013: (Almost) any code allowed in `constexpr` functions ver 2.0 Mutable Edition

However, the Committee decided that supporting loops (at least for) in `constexpr` methods is a must-have. In 2013 an amended version of the [N3597] Relaxing constraints on `constexpr` functions²⁴ proposal came out.

It described four ways to implement the "constexpr for" feature.

One of the choices was very far from the "general C++". It involved creating a completely new construction for iterations that would the `constexpr` code's functional style of the time. But that would have created a new sub language - the functional style `constexpr C++`.

²⁴<http://open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3597.html>

The choice closest to the "general C++" was not to replace quality with quantity. Instead, the idea was to try to support in `constexpr` a broad subset of C++ (ideally, all of it). **This option was selected.** This significantly affected `constexpr`'s subsequent history.

This is why there was a need for **object mutability** within `constexpr` evaluations. According to the proposal, an object created within a `constexpr` expression, can now be changed during the evaluation process - until the evaluation process or the object's lifetime²⁵ ends.

These evaluations still take place inside their "sandbox", nothing from the outside affects them. So, in theory, evaluating a `constexpr` expression with the same arguments will produce the same result (not counting the float- and double- calculation errors).

For a better understanding I copied a code snippet from the proposal:

```

1  constexpr int f(int a)
2  {
3      int n = a;
4      ++n;           // '++n' is not a constant expression
5      return n * a;
6  }
7
8  int k = f(4);      // OK, this is a constant expression.
9          // 'n' in 'f' can be modified because its lifetime
10         // began during the evaluation of the expression.
11
12 constexpr int k2 = ++k; // error, not a constant expression, cannot modify
13                     // 'k' because its lifetime did not begin within
14                     // this expression.
15 struct X
16 {
17     constexpr X() : n(5)
18     {
19         n *= 2;           // not a constant expression
20     }
21     int n;
22 };
23
24 constexpr int g()
25 {
26     X x;               // initialization of 'x' is a constant expression
27     return x.n;
28 }
29
30 constexpr int k3 = g(); // OK, this is a constant expression.
                           // 'x.n' can be modified because the lifetime of

```

²⁵<http://eel.is/c++draft/basic.life>

32 // 'x' began during the evaluation of 'g()'.

Let me note here, that at the time being the code below is compiled:

```

1 constexpr void add(X& x)
2 {
3     x.n++;
4 }
5
6 constexpr int g()
7 {
8     X x;
9     add(x);
10    return x.n;
11 }
```

Right now, a significant part of C++ can work within constexpr functions. Side effects are also allowed - if they are local within a constexpr evaluation. The constant evaluator became more complex, but still could handle the task.

5.11 2013: Legendary const methods and popular constexpr methods

The constexpr class member functions are currently automatically marked as const functions.

Proposal [N3598] constexpr member functions and implicit const²⁶ notices that it's not necessary to implicitly make the constexpr class member functions const ones.

This has become more relevant with mutability in constexpr evaluations. However, even before, this had been limiting the use of the same function in the constexpr and non-constexpr code:

```

1 struct B
2 {
3     constexpr B() : a() {}
4     constexpr const A &getA() const /*implicit*/ { return a; }
5     A &getA() { return a; } // code duplication
6     A a;
7 };
```

Interestingly, the proposal gave a choice of three options. The second option was chosen in the end:

1. Status quo. Cons: code duplication.
2. constexpr will not implicitly mean const. Cons: it breaks ABI²⁷—const is a part of the mangled method name.²⁸
3. Adding a new qualifier and writing constexpr A &getA() mutable return a;. Cons: a new buzzword at the end of the declaration.

²⁶<https://open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3598.html>

²⁷<https://cor3ntin.github.io/posts/abi/>

²⁸https://en.wikipedia.org/wiki/Name_mangling

5.12 2015-2016: Syntactic sugar for templates

In template metaprogramming, functions are usually overloaded if the body requires different logic depending on a type's properties. Example of scary code:

```

1 template <class T, class... Args>
2 enable_if_t<is_constructible_v<T, Args...>, unique_ptr<T>>
3 make_unique(Args&&... args)
4 {
5     return unique_ptr<T>(new T(forward<Args>(args)...));
6 }
7
8 template <class T, class... Args>
9 enable_if_t<!is_constructible_v<T, Args...>, unique_ptr<T>>
10 make_unique(Args&&... args)
11 {
12     return unique_ptr<T>(new T{forward<Args>(args)...});
13 }
```

Proposal [N4461] Static if resurrected²⁹ introduces the static_if expression (borrowed from the D language) to make code less scary:

```

1 template <class T, class... Args>
2 unique_ptr<T>
3 make_unique(Args&&... args)
4 {
5     static_if (is_constructible_v<T, Args...>)
6     {
7         return unique_ptr<T>(new T(forward<Args>(args)...));
8     }
9     else
10    {
11        return unique_ptr<T>(new T{forward<Args>(args)...});
12    }
13 }
```

This C++ fragment has a rather mediocre relation to constexpr expressions and works in a different scenario. But static_if in further revisions was renamed:

```

1 constexpr_if (is_constructible_v<T, Args...>)
2 {
3     return unique_ptr<T>(new T(forward<Args>(args)...));
4 }
5 constexpr_else
```

²⁹<http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4461.html>

```

6  {
7      return unique_ptr<T>(new T{forward<Args>(args)...});
8 }

```

Then some more renaming:

```

1 constexpr_if (is_constructible_v<T, Args...>)
2 {
3     return unique_ptr<T>(new T(forward<Args>(args)...));
4 }
5 constexpr_else
6 {
7     return unique_ptr<T>(new T{forward<Args>(args)...});
8 }

```

And the final version:

```

1 if constexpr (is_constructible_v<T, Args...>)
2 {
3     return unique_ptr<T>(new T(forward<Args>(args)...));
4 }
5 else
6 {
7     return unique_ptr<T>(new T{forward<Args>(args)...});
8 }

```

5.13 2015: Constexpr lambdas

A very good proposal, [N4487] Constexpr Lambda³⁰, works scrupulously through the use of the closure type in constexpr evaluations (and supported the forked Clang).

If you want to understand how it's possible to have constexpr lambdas, you need to understand how they work from the inside. There is an article about the history of lambdas³¹ that describes how proto-lambdas already existed in C++03. Today's lambda expressions have a similar class hidden deep inside the compiler.

5.13.1 [SPOILER BLOCK BEGINS]

5.13.1.1 Proto-lambda for [](int x) std::cout << x << std::endl;

```

1 #include <iostream>
2 #include <algorithm>
3 #include <vector>
4
5 struct PrintFunctor
6 {

```

³⁰<https://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4487.pdf>

³¹<https://www.cppstories.com/2019/02/lambdas-story-part1/?m=1>

```

7     void operator()(int x) const
8     {
9         std::cout << x << std::endl;
10    }
11 };
12
13 int main()
14 {
15     std::vector<int> v;
16     v.push_back(1);
17     v.push_back(2);
18     std::for_each(v.begin(), v.end(), PrintFunctor());
19 }
```

5.13.2 [[SPOILER BLOCK ENDS]]

If all the captured variables are literal types, then closure type is also proposed to be considered a literal type, and operator() is marked constexpr. The working example of constexpr lambdas:

```

1 constexpr auto add = [] (int n, int m)
2 {
3     auto L = [=] { return n; };
4     auto R = [=] { return m; };
5     return [=] { return L() + R(); };
6 };
7
8 static_assert(add(3, 4)() == 7, "");
```

5.14 2017-2019: Double standards

Proposal [P0595] The constexpr Operator³² considers the possibility of "knowing" inside the function where the function is being executed now - in a constant evaluator or in runtime. The author proposed calling constexpr() for this, and it will return true or false.

```

1 constexpr double hard_math_function(double b, int x)
2 {
3     if constexpr() && x >= 0)
4     {
5         // slow formula, more accurate (compile-time)
6     }
7     else
8     {
9         // quick formula, less accurate (run-time)
```

³²<https://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0595r0.html>

```

10    }
11 }
```

Then the operator was replaced with the "magic" function `std::is_constant_evaluated()` ([P0595R2])³³ and was adopted by the C++20 standard in this form.

If the proposal has been developed for a long time, then the authors sometimes do its "rebase" (similar to projects in git/svn), bringing it in line with the updated state.

Same thing here —the authors of [P1938] if consteval³⁴(I'll talk about consteval later) found that it's better to create a new entry:

```

1 if consteval { }
2 if (std::is_constant_evaluated()) { }
3 // ^^^ similar entries
```

This decision was made in C++23 —link to the vote.³⁵

5.15 2017-2019: We need to go deeper

In the `constexpr` functions during the `constexpr` evaluations we cannot yet use the debugger and output logs. Proposal [P0596] `std::constexpr_trace` and `std::constexpr_assert`³⁶ considers the introduction of special functions for these purposes.

The proposal was favorably accepted (link to the vote)³⁷ but has not yet been finalized

5.16 2017: The evil twin of the standard library

At this moment, `std::vector` (which is desirable to have in compile-time), cannot work in `constexpr` evaluations. It's mainly due to the unavailability of new/delete operators there.

The idea of allowing the new and delete operators into the constant evaluator looked too ambitious. Thus, a rather strange proposal [P0597] `std::constexpr_vector`³⁸ considers introducing the magic `std::constexpr_vector<T>`.

It is the opposite of `std::vector<T>` —can be created and modified only during `constexpr` evaluations.

```

1 constexpr constexpr_vector<int> x;           // Okay.
2 constexpr constexpr_vector<int> y{ 1, 2, 3 }; // Okay.
3 const constexpr_vector<int> xe;              // Invalid: not constexpr
```

It is not described how the constant evaluator should work with memory. @antoshkka and @ZaMaZaN4iK (the authors of many proposals) in [P0639R0] Changing attack vector of the `constexpr_vector` detected many cons of this approach. They proposed changing the work direction towards an abstract magic `constexpr` allocator that doesn't duplicate the entire standard library.

³³<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0595r2.html>

³⁴<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1938r0.html>

³⁵<https://github.com/cplusplus/papers/issues/677>

³⁶<https://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0596r0.html>

³⁷<https://github.com/cplusplus/papers/issues/602>

³⁸<http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0597r0.html>

5.17 2017-2019: Constexpr gains memory

The Constexpr ALL the thing!³⁹ presentation demonstrates an example of a constexpr library to work with JSON objects. The same thing, but in paper form, is in [P0810] constexpr in practice⁴⁰:

```

1  constexpr auto jsv
2   = R"({
3     "feature-x-enabled": true,
4     "value-of-y": 1729,
5     "z-options": {"a": null,
6                   "b": "220 and 284",
7                   "c": [6, 28, 496]}
8   })"_json;
9
10 if constexpr (jsv["feature-x-enabled"])
11 {
12   // code for feature x
13 }
14 else
15 {
16   // code when feature x turned off
17 }
```

The authors suffered greatly from the inability to use STL containers and wrote the std::vector and std::map analogues. Inside, these analogues have std::array that can work in constexpr.

Proposal [P0784] Standard containers and constexpr⁴¹ studies the possibility of inputting STL containers in constexpr evaluations.

Note. It's important to know what an allocator is. STL containers work with memory through it. What kind of an allocator —is specified through the tempte argument.⁴² If you want to get into the topic, read this article.⁴³

What's stopping us from allowing STL containers to be in constexpr evaluations? There are three problems:

1. Destructors cannot be declared constexpr. For constexpr objects it must be trivial.
2. Dynamic memory allocation/deallocation is not available.
3. placement-new is not available for calling the constructor in the allocated memory.

First problem. It was quickly fixed —the proposal authors discussed this problem with the developers of the MSVC++ frontend, GCC, Clang, EDG. The developers confirmed that the restriction can be relaxed. Now we can require from literal types to have a constexpr destructor, not the strictly trivial one.

Second problem. Working with memory is not very easy. The constant evaluator is obliged to catch

³⁹<https://youtu.be/HMB9oXFobJc>

⁴⁰<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0810r0.pdf>

⁴¹<http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p0784r7.html>

⁴²<https://en.cppreference.com/w/cpp/container/vector>

⁴³<https://habr.com/en/post/505632/>

undefined behavior in any form. If the constant evaluator finds undefined behavior, it should stop compilation.

This means that we should track not only objects, but also their "metadata" that keep everything in check and don't let us crash the program. A couple of examples of such metadata:

- Information about which field in union is active ([P1330])⁴⁴. An example of undefined behavior: writing to a member of inactive field.
- A rigid connection between a pointer or a reference and a corresponding previously created object. An example of undefined behavior: infinite set.

Because of this, it's pointless to use such methods:

```
1 void* operator new(std::size_t);
```

The reason is, there's no justification to bring `void*` to `T*`. In short, a new reference/pointer can either start pointing to an existing object or be created "simultaneously" with it.

That's why there are two options for working with memory that are acceptable in `constexpr` evaluations:

1. Simple new and delete expressions: `int* i = new int(42);`
2. Using a standard allocator: `std::allocator`⁴⁵ (it was slightly fixed).

Third problem. Standard containers separate memory allocations and the construction of objects in this memory. We figured out the problem with allocations —it is possible to provide it with a condition for metadata.

Containers rely on `std::allocator_traits`⁴⁶, for construction —on its `construct`⁴⁷ method. Before the proposal it has the following form:

```
1 template< class T, class... Args >
2 static void construct( Alloc& a, T* p, Args&&... args )
3 {
4     ::new (static_cast<void*>(p)) T(std::forward<Args>(args)...);
5     // ^~~ placement-new forbidden in constexpr evaluations
6 }
```

It cannot be used due to casting to `void*` and placement-new (forbidden in `constexpr` in general form). In the proposal it was transformed into

```
1 template< class T, class... Args >
2 static constexpr void construct( Alloc& a, T* p, Args&&... args )
3 {
4     std::construct_at(p, std::forward<Args>(args)...);
5 }
```

`std::construct_at`⁴⁸ is a function that works similarly to the old code in runtime (with a cast to `void*`). In `constexpr` evaluations:

⁴⁴<https://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1330r0.pdf>

⁴⁵<https://en.cppreference.com/w/cpp/memory/allocator>

⁴⁶https://en.cppreference.com/w/cpp/memory/allocator_traits

⁴⁷https://en.cppreference.com/w/cpp/memory/allocator_traits/construct

⁴⁸https://en.cppreference.com/w/cpp/memory/construct_at

The compiler constant evaluator will process it in a special way: apparently, by calling constructor from object connected to T*p.

It's enough to make it possible to use containers in constexpr evaluations.

At first, there were some restrictions on allocated memory. It should have been deallocated within the same constexpr evaluation without going beyond the "sandbox".

This new type of memory allocation is called transient constexpr allocations. Transient also means "temporal" or "short-lived".

The proposal also had a piece about non-transient allocation. It proposed releasing not all allocated memory. The unallocated memory "falls out" of the sandbox and would be converted to static storage —i.e. in the .rodata section. However, the committee considered this possibility "too brittle" for many reasons and has not accepted it yet.

The rest of the proposal was accepted.

5.18 2018: Catch me if you can

Proposal [P1002] Try-catch blocks in constexpr functions⁴⁹ brings try-catch blocks into constexpr evaluations.

This proposal is a bit confusing —throw was banned in constexpr evaluations at that moment. This means the catch code fragment never runs.

Judging by the document, this was introduced to mark all the std::vector functions as constexpr. In libc++ (STL implementation) a try-catch block is used in the vector::insert method.

5.19 2018: I said constexpr!

From personal experience I know the duality of the constexpr functions (can be executed at compile-time and runtime) leads to the fact that evaluations fall into runtime when you least expect it —code example⁵⁰. If you want to guarantee the right stage, you have to be creative —code example.⁵¹

Proposal [P1073] constexpr! functions⁵² introduces new keyword constexpr! for functions that should work only at compile-time. These functions are called immediate methods.

```

1  constexpr! int sqr(int n)
2  {
3      return n*n;
4  }
5
6  constexpr int r = sqr(100); // Okay.
7  int x = 100;
8  int r2 = sqr(x);           // Error: Call does not produce
                             // a constant.

```

⁴⁹<https://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1002r1.pdf>

⁵⁰<https://godbolt.org/z/f8xY7T9xn>

⁵¹<https://godbolt.org/z/9Prs41nhj>

⁵²<https://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1073r0.html>

If there's a possibility that variables unknown at the compilation stage may get into `constexpr!` (which is normal for `constexpr` functions), then the program won't compile:

```

1 constexpr! int sqrsqr(int n)
2 {
3     return sqr(sqr(n)); // Not a constant expression at this point,
4         // but that's okay.
5
6 constexpr int dblsqr(int n)
7 {
8     return 2 * sqr(n); // Error: Enclosing function is not
9         // constexpr!.
```

You cannot take a pointer/link to a `constexpr!` function. The compiler backend does not necessarily (and does not need to) know about the existence of such functions, put them in symbol tables, etc.

In further revisions of this proposal, `constexpr!` was replaced by `consteval`.

The difference between `constexpr!` and `consteval` is obvious. In the second case there's no fallbacks into runtime —example with `constexpr`⁵³; example with `consteval`.⁵⁴

5.20 2018: Too radical `constexpr`

At that moment a lot of proposals were about adding the `constexpr` specifier to various parts of the standard library. We do not discuss them in this article since it's the same template.

Proposal [P1235] Implicit `constexpr`⁵⁵ suggests marking all functions, that have a definition, as `constexpr`. But we can ban executing a function in compile-time:

1. <no specifier> —a method is marked by `constexpr`, if possible.
2. `constexpr` —works as it works now;
3. `constexpr(false)` —cannot be called at compile-time;
4. `constexpr(true)` —can be called only at compile-time, i.e. similar to `constexpr!/consteval`.

This proposal wasn't accepted —link to the vote.⁵⁶

5.21 2020: Long-lasting `constexpr` memory

As already discussed, after accepting proposal [P0784] Standard containers and `constexpr`⁵⁷, it became possible to allocate memory in `constexpr` evaluations. However, the memory must be freed before the end of a `constexpr` evaluation. These are so-called transient `constexpr` allocations.

Thus, you cannot create top-level `constexpr` objects of almost all STL containers and many other classes.

By "top-level object" I mean the result of the whole `constexpr` evaluation, for example:

⁵³<https://godbolt.org/z/f8xY7T9xn>

⁵⁴<https://godbolt.org/z/x6ds7vM8r>

⁵⁵<https://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1235r0.pdf>

⁵⁶<https://github.com/cplusplus/papers/issues/292>

⁵⁷<https://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p0784r7.html>

```

1 constexpr TFoo CalcFoo();
2 constexpr TFoo FooObj = CalcFoo();

```

Here the CalcFoo() call starts a constexpr evaluation, and FooObj - its result and a top-level constexpr object.

Proposal [P1974] Non-transient constexpr allocation using propconst⁵⁸ finds a way to solve the problem. To my mind, this is the most interesting proposal of all I gave in this article. It deserves a separate article. This proposal was given a green light and it's developing —a link to the ticket⁵⁹. I'll retell it here in an understandable form.

What's stopping us from having non-transient allocations? Actually, the problem is not to stuff chunks of memory into static storage (.bss/.rodata/their analogues), but to check that the whole scheme has a clear **consistency**.

Let's assume that we have a certain constexpr object. Its construction (more precisely, "evaluation") was provoked by non-transient allocations. This means that theoretical deconstruction of this object (i.e. calling its destructor) should release all non-transient memory. If calling the destructor would not release memory, then this is bad. There's no **consistency**, and a compilation error needs to be issued.

In other words, here's what a constant evaluator should do:

1. After seeing a request for a constexpr evaluation, execute it;
2. As a result of the evaluation, get an object that hides a bundle of constexpr variables of a literal type.
Also get a certain amount of unallocated memory (non-transient allocations);
3. Imitate a destructor call on this object (without actually calling it). Check that this call would release all non-transient memory;
4. If all checks were successful, then **consistency** proven. Non-transient allocations can be moved to static storage.

This seems logical and let's assume that it all was implemented. But then we'd get a problem with similar code with non-transient memory. The standard won't prohibit changing the memory and then checking for a destructor call will be pointless:

```

1 constexpr unique_ptr<unique_ptr<int>> uui
2     = make_unique<unique_ptr<int>>(make_unique<int>());
3
4 int main()
5 {
6     unique_ptr<int>& ui = *uui;
7     ui.reset();
8 }

```

Note. In reality, such code would be rebuffed by the OS for trying to write to a read-only RAM segment, but this is physical constancy. Code should have logical constancy.

Marking constexpr for objects entails marking them as const. All their members also become const.

However, if an object has a member of pointer type, it's bad —you won't be able to make it point to another object. But you can change the object to which it points.

⁵⁸<https://open-std.org/JTC1/SC22/WG21/docs/papers/2020/p1974r0.pdf>

⁵⁹<https://github.com/cplusplus/papers/issues/867>

Pointer types have two orthogonal constancy parameters:

1. Is it possible to start pointing to another object?
2. Is it possible to change the object pointed to?

In the end, we get 4 variants with different properties. OK —the string compiles, FAIL - it doesn't:

```

1 int dummy = 13;
2
3 int *test1 { nullptr };
4 test1 = &dummy; // OK
5 *test1 = dummy; // FAIL
6
7 int const *test2 { nullptr };
8 test2 = &dummy; // OK
9 *test2 = dummy; // FAIL
10
11 int * const test3 { nullptr };
12 test3 = &dummy; // FAIL
13 *test3 = dummy; // OK
14
15 int const * const test4 { nullptr };
16 test4 = &dummy; // FAIL
17 *test4 = dummy; // FAIL

```

”Normal” const leads to the third option, but constexpr needs the fourth one! I.e. it needs so-called deep-const.

The proposal based on a couple of old proposals suggests introducing new cv-qualifier⁶⁰ propconst (propagating const).

This qualifier will be used with pointer/reference types:

```

1 T propconst *
2 T propconst &

```

Depending on the T type, the compiler will either convert this word into const or delete it. The first case is if T is constant, the second if it's not.

```

1 int propconst * ---> int *
2 int propconst * const ---> int const * const

```

The proposal contains a table of propconst conversion in different cases:

Thus, the constexpr objects could acquire full logical consistency (deep-const):

```

1 constexpr unique_ptr<unique_ptr<int propconst> propconst> uui =
2     make_unique<unique_ptr<int propconst> propconst>(
3         make_unique<int propconst>()
4 );

```

⁶⁰<https://en.cppreference.com/w/cpp/language/cv>

```

5
6 int main()
7 {
8     // the two lines below won't compile
9     unique_ptr<int propconst>& ui1 = *uui;
10    ui1.reset();
11
12    // the line below compiles
13    const unique_ptr<int propconst>& ui2 = *uui;
14    // the line below won't compile
15    ui2.reset();
16 }
17
18 // P.S. This entry has not yet been adopted by the Committee.
19 // I hope they'll do better

```

5.22 2021: Constexpr classes

With the advent of fully constexpr classes, including std::vector, std::string, std::unique_ptr (in which all functions are marked as constexpr) there is a desire to say "mark all functions of the class as constexpr".

This makes proposal [P2350] constexpr class:⁶¹

```

1 class SomeType
2 {
3 public:
4     constexpr bool empty() const { /* */ }
5     constexpr auto size() const { /* */ }
6     constexpr void clear() { /* */ }
7     // ...
8 };
9 // ^^^ BEFORE
10
11 class SomeType constexpr
12 {
13 public:
14     bool empty() const { /* */ }
15     auto size() const { /* */ }
16     void clear() { /* */ }
17 // ...
18 };
19 // ^^^ AFTER

```

⁶¹<https://open-std.org/JTC1/SC22/WG21/docs/papers/2021/p2350r1.pdf>

I have an interesting story about this proposal. I didn't know about its existence and had an idea on stdcpp.ru⁶²to propose the same thing: a link to the ticket [RU]⁶³(which is not needed now).

Many almost identical proposals to the standard may appear almost simultaneously. This speaks in favor of the concept of multiple discovery⁶⁴: ideas are floating in the air and it doesn't matter who proposes them. If the community is big enough, the natural evolution occurs.

5.23 2019-∞: Constant interpreter in the compiler

constexpr evaluations can be very slow, because the constant evaluator on the syntax tree has evolved iteratively (starting with constant folding). Now the constant evaluator is doing a lot of unnecessary things that could be done more efficiently.

Since 2019, Clang has been developing ConstantInterpreter⁶⁵. In future it may replace constant evaluator in the syntax tree. It is quite interesting and deserves a separate article.

The idea of ConstantInterpreter is that you can generate bytecode on the base of a syntax tree and execute it on the interpreter. Interpreter supports the stack, call frames and a memory model (with metadata mentioned above).

The documentation for ConstantInterpreter is good. There are also a lot of interesting things in the video⁶⁶of the interpreter creator at the LLVM developers conference.

5.24 What else to look?

If you want to expand your understanding further, you can watch these wonderful talks from the experts. In each talk authors go beyond the story about constexpr. This may be constructing a constexpr library; a story about the use of constexpr in the future reflexpr⁶⁶; or the story about the essence of a constant evaluator and a constant interpreter.

- constexpr ALL the things!⁶⁷ Ben Deane& Jason Turner, C++Now 2017. A bit outdated but may be interesting. It's about building a constexpr library.
- Compile-time programming and reflection in C++20 and beyond⁶⁸, Louis Dionne, CppCon 2018. A lot of attention is paid to future reflection in C++.
- Useful constexpr⁶⁹by Antony Polukhin (@antoshkka⁷⁰), C++ CoreHard Autumn 2018. About compilers, reflection and metaclasses.
- The clang constexpr interpreter⁷¹, Nandor Licker, 2019 LLVM Developers' Meeting. Rocket science and a code interpreter for constexpr.

⁶²<https://stdcpp.ru/en/about>

⁶³<https://github.com/cpp-ru/ideas/issues/479>

⁶⁴https://en.wikipedia.org/wiki/Multiple_discovery

⁶⁵<https://clang.llvm.org/docs/ConstantInterpreter.html>

⁶⁶<https://youtu.be/LrggYD4aibg>

⁶⁶<https://en.cppreference.com/w/cpp/experimental/reflect>

⁶⁷<https://youtu.be/HMB9oXFobJc>

⁶⁸<https://youtu.be/CRDNPwXDVp0>

⁶⁹<https://youtu.be/MXEgTYDnfJU>

⁷⁰<https://habr.com/users/antoshkka>

⁷¹<https://youtu.be/LrggYD4aibg>

And here's also a link to a talk about a killer feature (in my opinion) [P1040] std::embed⁷², which would work great in tandem with constexpr. But, judging by the ticket⁷³, they plan to implement it in C++ something.

⁷²<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1040r3.html>

⁷³<https://github.com/cplusplus/papers/issues/28>

Chapter 6

C++20’s parenthesized aggregate initialization has some downsides

• Arthur O’ Dwyer  2022-06-03  ★★★

Somehow the topic of P0960 parenthesized aggregate initialization¹ has come up three times in the past week over on the cpplang Slack.² The good news is that usually the asker is curious why some reasonable-looking C++20 code fails to compile in C++17 —indicating that C++20’s new rules are arguably more intuitive than C++17’s.

But let’s start at the beginning.

6.1 The basics of brace-initialization versus parens-initialization

Regular readers of this blog may remember my simple initialization guidelines from “The Nightmare of Initialization in C++”³(2019-02-18):

- Use = whenever you can
- Use initializer-list syntax only for element initializers (of containers and aggregates).
- Use function-call syntax () to call a constructor, viewed as an object-factory⁴.

And one more rule:

- Every constructor should be **explicit**, unless you mean it to be usable as an implicit conversion.

So for example we can write

```
1 int a[] = {1, 2};  
2 std::array<int, 2> b = {1, 2};  
3 std::pair<int, int> p = {1, 2};  
4  
5 struct Coord { int x, y; };  
6 struct BadGrid { BadGrid(int width, int height); };
```

¹<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0960r3.html>

²<https://cppalliance.org/slack/>

³<https://quuxplusone.github.io/blog/2019/02/18/knightmare-of-initialization/>

⁴<https://quuxplusone.github.io/blog/2018/06/21/factory-vs-conversion/>

```

7  struct GoodGrid { explicit GoodGrid(int width, int height); };
8
9  Coord c = {1, 2};
10 auto g1 = BadGrid(10, 20);
11 auto g2 = GoodGrid(10, 20);

```

Notice that even when the type author of `BadGrid` has broken our rule that all constructors should be `explicit`, it's still possible for us as the client-code author to ignore that "mistake" and use function-call syntax () anyway. It would have been physically possible for us to write

```
1 BadGrid g1 = {10, 20}; // a grid with dimensions 10x20
```

but that would have been a solecism according to the guidelines I just laid out: The resulting `BadGrid` is not conceptually a sequence of two elements 10, 20 in the same way that an array, pair, or `Coord` is conceptually a sequence of two elements. Therefore, even though we can implicitly convert the braced initializer list 10, 20 to a `BadGrid`, we should not.

This guideline is particularly relevant when dealing with STL containers like `vector`, for three reasons:

- Every STL container does represent a sequence of elements.
- Every STL container has an absolutely massive constructor overload set.
- For historical reasons (with which I disagree), STL types deliberately make almost all of their constructors non-`explicit`.

Consider the following four initializations of `vector`:

```

1 std::vector<int> v1 = {10, 20};
2 std::vector<std::string> v2 = {"x", "y"};
3 auto v3 = std::vector<int>(10, 20);
4 auto v4 = std::vector<std::string>(10, "y");

```

These are all appropriate initializations according to my guidelines. The first two depict initializing a `vector` with a sequence of elements: 10, 20 or "x", "y". In both cases, the resulting vector ends up with 2 elements. The second two initialize a vector using one of its many "object-factory" constructors: specifically, the one that takes a size and a fill value. `v3` is initialized with 10 copies of 20; `v4` is initialized with 10 copies of "y". In case `v4`, we physically could have written

```
1 std::vector<std::string> v4 = {10, "y"}; // Faux pas!
```

but according to my guidelines we should not write that, for the simple reason that we're not intending to create a vector with the sequence of elements 10, "y". This situation is exactly analogous to the `BadGrid g1` case above.

6.2 Aggregate versus non-aggregate initialization

The next thing you should know about initialization in C++ is that C++ treats aggregates differently from non-aggregates. In C++98, this distinction was extremely visible, because curly braces could be used only to initialize aggregates, by which I mean plain old C-style structs and arrays:

```

1 Coord ca = Coord(1, 2); // syntax error in C++98
2 Coord cb = {1, 2};      // OK
3
4 BadGrid ga = BadGrid(1, 2); // OK
5 BadGrid gb = {1, 2};      // syntax error in C++98
6
7 std::pair<int, int> pa(1, 2); // OK
8 std::pair<int, int> pb = {1, 2}; // syntax error in C++98

```

In C++11 we got uniform initialization, which let us use to call constructors, like this:

```

1 Coord ca = Coord(1, 2); // still a syntax error in C++11
2 Coord cb = {1, 2};      // OK
3
4 BadGrid ga = BadGrid(1, 2); // OK
5 BadGrid gb = {1, 2};      // OK since C++11 (but solecism)
6
7 std::pair<int, int> pa(1, 2); // OK (but solecism)
8 std::pair<int, int> pb = {1, 2}; // OK since C++11 (in fact, preferred)

```

So in C++11 we have two different meanings for -initialization: Sometimes it means we’re calling a constructor with a certain set of arguments (or maybe with a `std::initializer_list`), and sometimes it means we’re initializing the members of an aggregate. The second case, aggregate initialization, has several special powers:

First, aggregate initialization initializes each member directly. When you call a constructor, the arguments are matched up to the types of the constructor parameters; when you do aggregate initialization, the initializers are matched up directly to the types of the object’s data members. This matters mainly for immovable types like `lock_guard`:

```

1 struct Agg {
2     std::lock_guard<std::mutex> lk_;
3 };
4 struct NonAgg {
5     std::lock_guard<std::mutex> lk_;
6     NonAgg(std::lock_guard<std::mutex> lk) : lk_(lk) {}
7 };
8
9 std::mutex m;
10 Agg a = { std::lock_guard(m) }; // OK
11 NonAgg na = { std::lock_guard(m) }; // oops, error

```

In the snippet above, the prvalue `std::lock_guard(m)` directly initializes `a.lk_`; but on the next line, the prvalue `std::lock_guard(m)` initializes only the constructor parameter `lk` —there’s no way for the author of that constructor to get `lk`’s value into the data member `na.lk_`, because `lock_guard` is immovable.

Even for movable types, the direct-initialization of aggregates can be a performance benefit. Recall

from “The surprisingly high cost of static-lifetime constructors”⁵(2018-06-26) that `std::initializer_list` is a view onto an immutable array. So:

```
1 std::string a[] = {"a", "b", "c"};
```

direct-initializes three `std::string` objects in an array `a`, whereas

```
1 std::vector<std::string> v = {"a", "b", "c"};
```

direct-initializes three `std::string` objects in an anonymous array, creates an `initializer_list<string>` referring to that array, and then calls vector’s constructor, which makes copies of those `std::strings`. Or again,

```
1 std::array<std::string, 3> b = {"a"s, "b"s, "c"s};
```

direct-initializes three `std::string` objects into the elements of `b` (which the Standard guarantees is an aggregate type), whereas

```
1 using S = std::string;
2 std::tuple<S, S, S> t = {"a"s, "b"s, "c"s};
```

direct-initializes three temporary `std::string` objects on the stack, and then calls tuple’s constructor with three `std::string&&s`. That constructor move-constructs into the elements of `t` and finally destroys the original temporaries.

Second, for backward-compatibility with C, aggregate initialization will value-initialize⁶ any trailing data members

```
1 struct sockaddr_in {
2     short           sin_family;
3     unsigned short  sin_port;
4     struct in_addr  sin_addr;
5     char            sin_zero[8];
6 };
7
8 sockaddr_in s = {AF_INET};
9 assert(s.sin_port == 0);
10 assert(s.sin_addr.s_addr == 0);
11 assert(s.sin_zero[7] == 0);
```

This is useful for C compatibility, but it does lead to some surprising results: whether it’s “OK” to omit the initializers of trailing elements depends on whether the type being initialized is an aggregate or not, even though we’re uniformly using braced-initializerlist initialization syntax in both cases.

```
1 Coord c = {42};      // OK: Coord is an aggregate
2 BadGrid g = {42};    // ill-formed: BadGrid is not an aggregate
3
4 std::array<int,3> b = {1, 2};      // OK: array is an aggregate
5 std::tuple<int,int,int> t = {1, 2}; // ill-formed: tuple is not an aggregate
```

⁵<https://quuxplusone.github.io/blog/2018/06/26/cost-of-static-lifetime-constructors/>

⁶<https://eel.is/c++draft/dcl.init.general#def:value-initialization>

6.3 The coincidence of zero-argument initialization

Because C++11 introduced uniform initialization, T is permitted to call T's default constructor if it has one; or, if T is an aggregate, then T will do aggregate initialization where every member is value-initialized.

```

1  using B = std::array<int,3>;
2  using T = std::tuple<int,int,int>;
3
4  B b = B{}; // value-initialize, i.e., {0,0,0}
5  T t = T{}; // call the zero-argument constructor, i.e., {0,0,0}
```

Also, ever since C++98, there's been wording⁷ to make T() do value-initialization as a special case. (Sure, it looks like we're calling T's zero-argument constructor—and you can totally think of it that way in practice—but technically this is a special-case syntax for value-initializing⁸ T, which in turn calls T's default constructor if it has one, but otherwise recursively value-initializes T's members.)

```

1  B b = B(); // value-initialize, i.e., {0,0,0}
2  T t = T(); // value-initialize, i.e. call the default constructor, i.e., {0,0,0}
```

6.4 emplace_back presents a stumbling block

C++11 also introduced the idea of perfect-forwarding arguments through APIs like `emplace_back`. The idea of `emplace_back` is that you pass in some arguments `Args&&...` args, and then the vector will construct its new element using a placement-new expression like `::new (p) T(std::forward<Args>(args))`. Notice the parentheses there—not braces! This is important because we want to ensure that we get consistent behavior when emplacing an STL container. Compare the following snippet with our first section's v3/v4 example:

```

1  std::vector<std::vector<int>> vvi;
2  vvi.emplace_back(10, 20); // emplace vector<int>(10, 20)
3
4  std::vector<std::vector<std::string>> vvs;
5  vvs.emplace_back(10, "y"); // emplace vector<string>(10, "y")
```

But now, consider this C++17 code:

```

1  using T = std::tuple<int,int,int>; // non-aggregate
2  using B = std::array<int,3>; // aggregate
3
4  std::vector<T> vt; // vector of non-aggregates
5  std::vector<B> vb; // vector of aggregates
6
7  vt.emplace_back(); // OK, emplaces T()
8  vb.emplace_back(); // 1: OK, emplaces B()
```

⁷<https://timsong-cpp.github.io/cppwp/n3337/expr.type.conv#2>

⁸<https://eel.is/c++draft/dcl.init.general#def.value-initialization>

```

10 vt.emplace_back(1,2,3); // OK, emplaces T(1,2,3)
11 vb.emplace_back(1,2,3); // 2: Error in C++17!

```

Line 1 emplaces an `array` object constructed with `B()`, which, as we saw in the preceding section, means value-initialization: the same as 0,0,0.

Line 2 attempts to emplace an `array` constructed with `B(1,2,3)` —and this fails, because `B` has no constructor taking three ints!

Whether a given `emplace_back` is legal depends on whether the type being initialized is an aggregate or not, even though we’re using the same `emplace_back` syntax in both cases.

6.5 C++20’s solution: Parens-init for aggregates

C++20 addressed the above `emplace_back` quirk by saying: well, if the problem is that `B(1,2,3)` isn’t legal syntax, let’s just make it legal! C++20 adopted P0960 “Allow initializing aggregates from a parenthesized list of values,”⁹ which extends the rules of initialization to cover the case where “the destination type is a (possibly cv-qualified) aggregate class `A` and the initializer is a parenthesized expression-list.” In that case, initialization proceeds just as in the curly-brace case, omitting a few minor quirks that are triggered (since C++11) by the curly-brace syntax specifically:

- Curly-braced initializers are evaluated strictly left-to-right; parenthesized initializers can be evaluated in any order.
- Curly-braced initializers forbid narrowing conversions (such as double-to-int); parenthesized initializers do not.
- Curly-braced prvalues bound to reference data members can be lifetime-extended; parenthesized prvalues are never lifetime-extended.
- Curly-braced initialization lists can involve brace elision (e.g. depicting 1, 2, 3, 4 as 1, 2, 3, 4); parenthesized initialization lists cannot.
- Curly-braced initialization lists can include C++20 designated initializers¹⁰; parenthesized initialization lists cannot.

The result is that the following is legal C++20 (but not legal C++17):

```

1 struct Coord { int x, y; }; // aggregate
2 std::vector<Coord> vc;
3 vc.emplace_back();          // OK since C++11, emplaces Coord() i.e. {0, 0}
4 vc.emplace_back(10, 20);    // OK since C++20, emplaces Coord(10, 20) i.e. {10, 20}

```

6.6 The unintended consequences

This being C++, naturally there are some rough edges and pitfalls. I count at least three.

First, remember that even though `vc.emplace_back(10,20)` emplaces the equivalent of 10,20, it doesn’t actually use curly braces! `emplace_back` still uses the round-parens syntax `Coord(10,20)`, and simply relies on aggregate parens-init to convert that into the equivalent of 10,20. For non-aggregate types, where `T(x,y)` and `Tx,y` do different things, you’re still going to get the `T(x,y)` behavior!

⁹<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0960r3.html>

¹⁰<https://eel.is/c++draft/dcl.init.general#nt:designated-initializer-list>

```

1 std::vector<std::vector<int>> vvi;
2 vvi.emplace_back(10, 20); // OK since C++11, emplaces vector<int>(10, 20)
3 assert(vvi[0].size() == 10); // not 2

```

Second, look back at that list of curly-brace quirks ignored by aggregate parens-init. Notice one quirk that's not on that list: both kinds of aggregate initialization are allowed to omit trailing initializers!

```

1 Coord c1 = {10}; // OK since C++98, equivalent to {10, 0}
2 Coord c2 = Coord(10); // OK since C++20, equivalent to {10, 0}
3 vc.emplace_back(10); // OK since C++20, emplaces Coord(10) i.e. {10, 0}

```

In fact, since `Coord(10)` is legal in C++20, we can even write

```
1 auto c3 = static_cast<Coord>(10); // OK since C++20, equivalent to {10, 0}
```

which strikes me as probably terrible.

Vice versa, even though C++20 made it legal to aggregate-initialize array types with parentheses in general, some of the syntactic space was deliberately reserved for future standardization. For example, parens-init works for array variables but not array prvalues:

```

1 using A = int[3];
2 A a1(1,2,3); // OK since C++20
3 auto a2 = A(1,2,3); // still an error!

```

Third, probably the most annoying pitfall (although it does satisfactorily confirm my prejudice against `std::array`¹¹): parens-init is basically useless for `std::array`!

```

1 using T = std::tuple<int, int, int>; // non-aggregate
2 using B = std::array<int, 3>; // aggregate
3 std::vector<T> vt;
4 std::vector<B> vb;
5 vt.push_back({1,2,3}); // OK since C++11
6 vb.push_back({1,2,3}); // OK since C++11
7
8 vt.emplace_back(1,2,3); // OK since C++11
9 vb.emplace_back(1,2,3); // still an error!

```

The reason is that `B(1,2,3)` remains ill-formed even in C++20; and the reason for that is the fourth quirk in the list above: parenthesized initializer lists don't do brace elision the way braced initializer lists do! `std::array<T,N>` is guaranteed¹² to be an aggregate initializable with N elements of type T, but nothing about the data's underlying "shape" is guaranteed: does it have N data members of type T? or one data member of type `T[N]`? or $N/2$ data members of type `T[2]`? Brace elision allows us not to care about such details when we use curly-braced aggregate initialization, but not when we use parenthesized aggregate initialization.

```

1 B b1(1,2,3); // still an error! (technically implementation-defined)
2 B b2{{1,2,3}}; // OK since C++11; direct-initializes (technically implementation-def
3 B b3({1,2,3}); // OK since C++11; move-constructs
4 vb.emplace_back({1,2,3}); // still an error!

```

¹¹<https://quuxplusone.github.io/blog/2020/08/06/array-size/>

¹²<https://eel.is/c++draft/array.overview#2>

The above b3 is already legal in C++11; it represents direct-initialization of a B from 1,2,3 (which overload resolution will decide to treat as B1,2,3), and C++20 doesn't change this. And `vb.emplace_back(1,2,3)` remains ill-formed, because `emplace_back`'s perfect forwarding wants its `Args&&...` to have nicely deducible types, and a brace-enclosed initializer list like 1,2,3 has no type.

6.7 The bottom line

C++20's parenthesized aggregate initialization solves the tipmost link in a rather long chain of unintended consequences:

- Aggregate types are quietly treated differently from non-aggregates; for example, they get more-direct initialization.
- `emplace_back` uses parens-initialization rather than brace-initialization, and we can't change that without breaking `vvi.emplace_back(10, 20)`.
- Therefore `vc.emplace_back(10, 20)` wants to parens-initialize `Coord(10, 20)`; in C++17 that's ill-formed unless we add a constructor `Coord(int, int)`.
- But giving a type a constructor makes it a non-aggregate, which means it loses the performance benefits of aggregate initialization. (Not that `Coord` cares about those benefits.)
- C++20 makes `vc.emplace_back(10, 20)` legal, by permitting `Coord(10, 20)` to do aggregate initialization just like `Coord{10, 20}`.

But as a side effect, C++20 makes `static_cast<Coord>(10)` legal (surprise!); and even after all this rigmarole, C++20 fails to make either `vb.emplace_back(1,2,3)` or `vb.emplace_back(1,2,3)` work as expected. That's bad.

But I'll leave you by reiterating the good news: When I've seen people bring up this topic on Slack¹³, they're usually not at all surprised that both of these lines

```

1 std::vector<Coord> vc;
2 vc.push_back({10, 20}); // A, OK since C++11
3 vc.emplace_back(10, 20); // B, OK since C++20

```

both work in C++20. Generally, they're surprised only that line B failed to work before C++20! So, in at least that narrow sense, C++20's parenthesized aggregate initialization has improved the story for newcomers to C++.

¹³<https://cppalliance.org/slack/>

Chapter 7

(Non) Static Data Members Initialization, from C++11 till C++20

👤 Bartłomiej Filipek 📅 2022-07-11 💬 ★★★

With Modern C++ and each revision of the Standard, we get more comfortable ways to initialize data members. There's non-static data member initialization (from C++11) and inline variables (for static members since C++17).

In this blog post, you'll learn how to use the syntax and how it has changed over the years. We'll go from C++11, through C++14, and C++17 until C++20.

Updated in July 2022: added more examples, use cases, and C++20 features.

7.1 Initialisation of Data members

Before C++11, if you had a class member, you could only initialize it with a default value through the initialization list in a constructor.

```
1 // pre C++11 class:  
2 struct SimpleType {  
3     int field;  
4     std::string name;  
5     SimpleType() : field(0), name("Hello World") {}  
6 }
```

Since C++11, the syntax has been improved, and you can initialize field and name in the place of the declaration:

```
1 // since C++11:  
2 struct SimpleType {  
3     int field = 0; // works now!  
4     std::string name { "Hello World " } // alternate way with {}  
5 }
```

```

6     SimpleType() { }
7 }
```

As you can see, the variables get their default value in the place of declaration. There's no need to set values inside a constructor.

The feature is called **non-static data member initialization**, or NSDMI in short.

What's more, since C++17, we can initialise static data members thanks to inline variables:

```

1 struct OtherType {
2     static const int value = 10;
3     static inline std::string className = "Hello Class";
4
5     OtherType() { }
6 }
```

There's no need to define className in a corresponding cpp file. The compiler guarantees that all compilation units will see only one definition of the static member. Previously, before C++17, you had to put the definition in one of cpp files.

Please note that for constant integer static fields (value), we could initialize them “in place” even in C++98.

Let's explore those useful features: NSDMI and inline variables. We'll see the examples and how the features improved over the years.

7.2 NSDMI - Non-static data member initialization

In short, the compiler performs the initialization of your fields as you'd write it in the constructor initializer list.

```
1 SimpleType() : field(0) { }
```

Let's see this in detail:

7.3 How It works

With a bit of “machinery,” we can see when the compiler performs the initialization.

Let's consider the following type:

```

1 struct SimpleType
2 {
3     int a { initA() };
4     std::string b { initB() };
5
6     // ...
7 };
```

The implementation of `initA()` and `initB()` functions have side effects and they log extra messages::

```

1 int initA() {
2     std::cout << "initA() called\n";
3     return 1;
4 }
5
6 std::string initB() {
7     std::cout << "initB() called\n";
8     return "Hello";
9 }
```

This allows us to see when the code is called.

For example:

```

1 struct SimpleType
2 {
3     int a { initA() };
4     std::string b { initB() };
5
6     SimpleType() { }
7     SimpleType(int x) : a(x) { }
8 };
```

And the use:

```

1 std::cout << "SimpleType t10\n";
2 SimpleType t0;
3 std::cout << "SimpleType t1(10)\n";
4 SimpleType t1(10);
```

The output:

```

1 SimpleType t0:
2 initA() called
3 initB() called
4 SimpleType t1(10):
5 initB() called
```

t0 is default initialized; therefore, both fields are initialized with their default value.

In the second case, for **t1**, only one value is default initialized, and the other comes from the constructor parameter.

As you might already guess, the compiler performs the initialization of the fields as if the fields were initialized in a “member initialization list.” So they get the default values before the constructor’s body is invoked.

In other words the compiler expands the code:

```

1 int a { initA() };
2 std::string b { initB() };
```

```

3
4 SimpleType() { }
5 SimpleType(int x) : a(x) { }

into

1 int a;
2 std::string b;

3
4 SimpleType() : a(initA()), b(initB()) { }
5 SimpleType(int x) : a(x), b(initB()) { }

```

How about other constructors?

7.4 Copy and Move Constructors

The compiler initializes the fields in all constructors, including copy and move constructors. However, when a copy or move constructor is default, there's no need to perform that extra initialization.

See the examples:

```

1 struct SimpleType
2 {
3     int a { initA() };
4     std::string b { initB() };
5
6     SimpleType() { }
7
8     SimpleType(const SimpleType& other) {
9         std::cout << "copy ctor\n";
10
11         a = other.a;
12         b = other.b;
13     };
14
15 };

```

And the use case:

```

1 SimpleType t1;
2 std::cout << "SimpleType t2 = t1:\n";
3 SimpleType t2 = t1;

```

The output:

```

1 SimpleType t1:
2 initA() called
3 initB() called

```

```

4 SimpleType t2 = t1:
5 initA() called
6 initB() called
7 copy ctor

```

See code here [@Wandbox](#).

The compiler initialized the fields with their default values in the above example. That's why it's better also to use the initializer list inside a copy constructor:

```

1 SimpleType(const SimpleType& other) : a(other.a), b(other.b) {
2     std::cout << "copy ctor\n";
3 }

```

We get:

```

1 SimpleType t1:
2 initA() called
3 initB() called
4 SimpleType t2 = t1:
5 copy ctor

```

The same happens if you rely on the copy constructor generated by the compiler:

```

1 SimpleType(const SimpleType& other) = default;

```

You get a similar behavior for the move constructor.

7.5 Other forms of NSDMI

Let's try some other examples and see all options that we can initialize a data member using NSDMI:

```

1 struct S {
2     int zero {};           // fine, value initialization
3     int a = 10;            // fine, copy initialization
4     double b { 10.5 };    // fine, direct list initialization
5     // short c ( 100 );   // err, direct initialization with parens
6     int d { zero + a }; // dependency, risky, but fine
7     // double e { *mem * 2.0 }; // undefined!
8     int* mem = new int(d);
9     long arr[4] = { 0, 1, 2, 3 };
10    std::array<int, 4> moreNumbers { 10, 20, 30, 40 };
11    // long arr2[] = { 1, 2 }; // cannot deduce
12    // auto f = 1;          // err, type deduction doesn't work
13    double g { compute() };
14
15    ~S() { delete mem; }
16    double compute() { return a*b; }
17 }

```

See [@Compiler Explorer](#).

Here's the summary:

- `zero` uses value initialization, and thus, it will get the value of 0,
- `a` uses copy initialization,
- `b` uses direct list initialization,
- `c` would generate an error as direct initialization with parens is not allowed for NSDMI,
- `d` initializes by reading `zero` and `a`, but since `d` appears later in the list of data members, it's okay, and the order is well defined,
- `e`, on the other hand, would have to read from a data member `mem`, which might not be initialized yet (since it's further in the declaration order), and thus this behavior is undefined,
- `mem` uses a memory allocation which is also acceptable,
- `arr[4]` declares and initializes an array, but you need to provide the number of elements as the compiler cannot deduce it (as in `arr2`),
- similarly we can use `std::array<type, count>` for `moreNumbers`, but we need to provide the count and the type of the array elements,
- `f` would also generate an error, as `auto` type deduction won't work,
- `g` calls a member function to compute the value. The code is valid only when that function calls reads from already initialized data members.

7.6 C++14 Updates for aggregates, NSDMI

Originally, in C++11, if you used default member initialisation then your class couldn't be an aggregate type:

```
1 struct Point { float x = 0.0f; float y = 0.0f; };
2 // won't compile in C++11
3 Point myPt { 10.0f, 11.0f};
```

I was unaware of this issue, but Shafik Yaghmour noted that in the comments below the article.

In C++11, spec did not allow aggregate types to have such initialization, but in C++14 this requirement was removed. Link to the StackOverflow question ¹ with details

Fortunately, it's fixed in C++14, so

```
1 Point myPt { 10.0f, 11.0f};
```

Compiles as expected; see [@Wandbox](#)

7.7 C++20 Updates for bit fields

Since C++11, the code only considered "regular" fields...but how about bit fields in a class?

¹<https://stackoverflow.com/questions/27118535/c11-aggregate-initialization-for-classes-with-non-static-member-initializers>

```

1 class Type {
2     int value : 4;
3 };

```

This is only a recent change in C++20 that allows you to write:

```

1 class Type {
2     int value : 4 = 0;
3     int second : 4 { 10 };
4 };

```

The proposal was accepted into C++20 as Default Bit Field Initialiser for C++20 P0683.²

7.8 The case with auto

Since we can declare and initialize a variable inside a class, there's an interesting question about `auto`. Can we use it? It seems quite a natural way and would follow the AAA (Almost Always Auto) rule.

You can use `auto` for static variables:

```

1 class Type {
2     static inline auto theMeaningOfLife = 42; // int deduced
3 };

```

But not as a class non-static member:

```

1 class Type {
2     auto myField { 0 }; // error
3     auto param { 10.5f }; // error
4 };

```

Unfortunately, `auto` is not supported. For example, in GCC I get

```
1 error: non-static data member declared with placeholder 'auto'
```

While static members are just static variables, and that's why it's relatively easy for the compiler to deduce the type, it's not that easy for regular members. This is mostly because of the possible cyclic dependencies of types and the class layout. If you're interested in the full story, you can read this great explanation at cor3ntin blog: The case for Auto Non-Static Data Member Initializers | cor3ntin.³

7.9 The case with CTAD - Class Template Argument Deduction

Similarly, as with `auto` we also have limitations with non-static member variables and CTAD:
It works for static variables

```

1 class Type {
2     static inline std::vector ints { 1, 2, 3, 4, 5, 6, 7}; // deduced vector<int>
3 };

```

²<https://wg21.link/P0683>

³https://cor3ntin.github.io/posts/auto_nsmdi/

But not as a non-static-member:

```
1 class Type {
2     std::vector ints { 1, 2, 3, 4, 5, 6, 7}; // error!
3 };
```

On GCC 10.0 I get

```
1 error: 'vector' does not name a type
```

7.10 Advantages of NSDMI

- It's easy to write.
- You can be sure that each member is initialized correctly.
- The declaration and the default value are in the same place, so it's easier to maintain.
- It's much easier to conform to the rule that every variable should be initialized.
- It is beneficial when we have several constructors. Previously, we would have to duplicate the initialization code for members or write a custom method, like `InitMembers()`, that would be called in the constructors. Now, you can do a default initialization, and the constructors will only do their specific jobs.

7.11 Any negative sides of NSDMI?

On the other hand, the feature has some limitations and inconveniences:

- Using NSDMI makes a class not trivial, as the default constructor (compiler-generated) has to perform some work to initialize data members.
- Performance: When you have performance-critical data structures (for example, a Vector3D class), you may want to have an “empty” initialization code. You risk having uninitialized data members, but you might save several CPU instructions.
- (Only until C++14) NSDMI makes a class non-aggregate in C++11. Thanks, Yehezkel, for mentioning that! This drawback also applies to static variables that we'll discuss later.
- They have limitations in the case of `auto` type deduction and CTAD, so you need to provide the type of the data member explicitly.
- You cannot use direct initialization with parens, to fix it, you need list initialization or copy initialization syntax for data members.
- Since the default values are in a header file, any change can require recompiling dependent compilation units. This is not the case if the values are set only in an implementation file.
- Might be hard to read if you rely on calling member functions or depend on other data members.

Do you see any other issues?

7.12 Inline Variables C++17

So far, we have discussed non-static data members. Do we have any improvements for declaring and initializing static variables in a class?

In C++11/14, you had to define a variable in a corresponding cpp file:

```

1 // a header file:
2 struct OtherType {
3     static int classCounter;
4
5     // ...
6 };
7 // implementation, cpp file
8 int OtherType::classCounter = 0;
```

Fortunately, with C++17, we also got **inline variables**, which means you can define a `static inline` variable inside a class without defining them in a cpp file.

```

1 // a header file, C++17:
2 struct OtherType {
3     static inline int classCounter = 0;
4
5     // ...
6 };
```

One note: before C++17, you could declare and define a constant static integer data member, but since C++17 it's "extended" to all types (and also mutable) through the `inline` keyword.

```

1 // a header file, C++17:
2 struct MyClass {
3     static const int ImportantValue = 99; // declaration and definition in one place
4
5     // ...
6 };
```

The compiler guarantees that there's precisely one definition of this static variable for all translation units, including the class declaration. Inline variables are still static class variables so that they will be initialized before the `main()` function is called (You can read more in my separate article What happens to your static variables at the start of the program?⁴).

The feature makes it much easier to develop header-only libraries, as there's no need to create cpp files for static variables or use some hacks to keep them in a header file.

Here's the full example at [@Wandbox](#)

7.13 Summary

In this article, we reviewed how in-class member initialization changed with Modern C++.

In C++11, we got NSDMI - non-static data member initialization. You can now declare a member variable and init that with a default value. The initialization will happen before each constructor body is called, in the constructor initialization list.

⁴<https://www.cppstories.com/2018/02/staticvars/>

NSDMI improved with C++14 (aggregates) and in C++20 (bit fields are now supported).

The feature is also reflected in C++ Core Guidelines:

C.48: Prefer in-class initializers to member initializers in constructors for constant initializers⁵

Reason: Makes it explicit that the same value is expected to be used in all constructors. Avoids repetition. Avoids maintenance problems. It leads to the shortest and most efficient code. Here's a "summary" example that combines the features:

What's more, in C++17, we got **inline variables**, which means you can declare and initialize a static member without the need to do that in a corresponding .cpp file.

Here's a "summary" example that combines the features:

```

1 struct Window
2 {
3     inline static unsigned int default_width = 1028;
4     inline static unsigned int default_height = 768;
5
6     unsigned int _width { default_width };
7     unsigned int _height { default_height };
8     unsigned int _flags : 4 { 0 };
9     std::string _title { "Default Window" };
10
11    Window() { }
12    Window(std::string title) : _title(std::move(title)) { }
13    // ...
14};

```

Play at [@Wandbox](#)

For simplicity, `default_width` and `default_height` are static variables that can be loaded, for example, from a configuration file, and then be used to initialize a default Window state.

Your Turn

- Do you use NSDMI in your projects?
- Do you use static Inline variables as class members?

7.14 Extra Links

- “Embracing Modern C++ Safely” by J. Lakos, V. Romeo , R. Khlebnikov, A. Meredith, a wonderful and very detailed book about latest C++ features, from C++11 till C++14 in the 1st edition,⁶
- “Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14” by Scott Meyers.
- Bjarne Stroustrup C++ FAQ : In-class member initializers.⁷

⁵<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c48-prefer-in-class-initializers-to-member-initializers-in-constructors-for-constant-initializers>

⁶<https://www.i-programmer.info/bookreviews/5-c/15251-embracing-modern-c-safely.html>

⁷<https://www.stroustrup.com/C++11FAQ.html#member-init>

- VC++ Blog: The Future of Non-Static Data Member Initialization.⁸
- Core C++ 2019: Initialisation in modern C++ by Timur Doumler.⁹
- CppCon 2018: “The Nightmare of Initialization in C” by Nicolai Josuttis.¹⁰
- CppCon 2021: Back To Basics: The Special Member Functions by Klaus Iglberge¹¹
- ACCU 2022: What Classes We Design and How - by Peter Sommerlad.¹²

⁸<https://devblogs.microsoft.com/cppblog/the-future-of-non-static-data-member-initialization/>

⁹<https://www.youtube.com/watch?v=v0jM4wm1zYA>

¹⁰<https://www.youtube.com/watch?v=7DTIWPGX6zs>

¹¹<https://www.youtube.com/watch?v=9BM5LAvNtus>

¹²<https://www.youtube.com/watch?v=fzsBZicBe88>

Chapter 8

[C++] Static, Dynamic Polymorphism, CRTP and C++20's Concepts

👤 Thomasj 📅 2022-09-24 💬 ★★

Let's talk about Polymorphism, CRTP as a way to use static polymorphism and how C++20 can change the way how we write code.

8.1 Dynamic Polymorphism

When it comes to interfaces, we think of a base class and a concrete implementation. With dynamic polymorphism we can write this code:

```
1 // base class contains a virtual function
2 struct base {
3     virtual void do_work() = 0;
4 };
5
6 // which we override, to provide a concrete implementation
7 struct derived : public base {
8     void do_work() override {
9         // let's do some work here ...
10    }
11 }
```

In general, there's nothing wrong to do so in the first place, but in a C++ context there are some comments here.

First, since we use virtual functions we create a vtable and during runtime we call the appropriate function. There are benchmarks and articles which show that this has impact on the applications performance.

Second, do we really need to take the concrete implementation during runtime? In my experience it's more often the case that I already know during compile time which one I need. I'm often confused why I use something dynamic when I don't need it dynamic.

8.2 Static Polymorphism

Let's move the implementation from above to a static one. CRTP (Curiously recurring template pattern) is a method which can be seen as static polymorphism (note, there are other advantages to use CRTP it's not that CRTP is static polymorphism).

```

1 // now we have a templated base class
2 template<typename T>
3 struct base {
4     // no virtual function here
5     void do_work () {
6         // and we cast this to the template type, where the actual implementation lives
7         static_cast<T*>(this)->do_work_impl();
8     }
9 };
10
11 // our class derived inherits from base and passes itself as template parameter
12 class derived : public base<derived> {
13     // declare base as friend so do_work_impl can be called
14     // because we still want to call the base class function
15     friend class base<derived>;
16     void do_work_impl() {
17         // let's do some work here ...
18     }
19 };

```

We'll make sure that base is used that way and therefore with `static_cast<T*>(this)` we have access to the actual concrete implementation. Now we have a static approach of an interface.

... But:

- It's not so clear what methods needs to be implemented (compared to virtual ones)
- We need to rename the functions on the concrete implementation

8.3 C++20's Concepts

Let's move on and introduce concepts, which are basically named constraints in our code. Some general examples of concepts are:

```

1 // we can assign a condition to a concept
2 // in this case whether my given type is an integral type or not
3 template<typename T>
4 concept is_numeric = std::is_integral_v<T>;
5
6
7 // there are different ways to apply the concept

```

```

8
9 //1. use requires before the function
10 template<typename T>
11 requires is_numeric<T>
12 void foo(T t) {
13     //...
14 }
15
16 //2. use requires after the function
17 template<typename T>
18 void foo(T t) requires is_numeric<T> {
19     //...
20 }
21
22 //3. use the concept as template argument
23 template<is_numeric T>
24 void foo(T t) {
25     // ...
26 }
27
28 //4. use the concept in combination with auto as argument
29 void foo(is_numeric auto t) {
30     // ...
31 }
```

And we also can have sort of code as concept, which is evaluated at compile time. Consider a function which needs to increment (post and pre) a variable. You can write concepts like this:

```

1 template<typename T>
2 // introduce requires keyword where we pass a type T in
3 concept has_increment = requires(T t)
4 // all the code is validated at compile time
5 // if there would be errors, this would not compile
6 {
7     t++;
8     ++t;
9 };
10
11
12 template<has_increment T>
13 void foo(T t){
14     // ...
15 }
```

```

17 struct bar {};
18
19 //...
20
21 foo(1); // ok
22 foo(bar{}); // error

```

And we have another benefit here, which is clear error messages. With the last example we'd get this error:

```

1 <source>:91:9: required for the satisfaction of 'has_increment<T>' [with T = baz]
2 <source>:91:25: in requirements with 'T t' [with T = baz]
3 <source>:92:6: note: the required expression '(t++)' is invalid
4   92 |     t++;
5       |     ~~~
6 <source>:93:5: note: the required expression '++ t' is invalid
7   93 |     ++t;
8       |     ^~~

```

And now we can apply concepts to create an interface for our base class where we want to implement `do_work()`:

```

1 // we create a concept can_work to check if do_work is implemented
2 // this will describe our interface
3 template <typename T>
4   concept can_work = requires(T t) {
5     t.do_work();
6   };
7
8 // now we apply this concept to an empty type which represents a worker (or our base class)
9 template<can_work T>
10 struct worker : public T {};
11
12 // now create a concrete worker (corresponding derived) where we implement do_work
13 struct concrete_worker {
14   void do_work() {
15     // ...
16   }
17 };
18
19 // nice to have: an alias for our concrete worker
20 using my_worker = worker<concrete_worker>;
21
22 //...
23 // which we can use now
24 my_worker w;
25 w.do_work();

```

And there we have it, we expressed our interface in the concept and on the concrete implementation we removed the inheritance. With the final alias we can then use the concrete type we want.

8.4 Conclusion

I really like the idea of concepts. This will affect the way we used to write code, especially generic code. It makes it easier to write and to read. Also the error messages way better. Compared to templates, which make it often hard to understand and comprehend this can be a time safer.

But that’s it for now. Start using concepts.

Best Thomas

Chapter 9

Structured bindings in C++17, 5 years later

▀ Bartłomiej Filipek 📅 2022-12-19 🔮 ★★★

Structured bindings are a C++17 feature that allows you to bind multiple variables to the elements of a structured object, such as a tuple or struct, in a single declaration. This can make your code more concise and easier to read, especially when working with complex data structures. In this blog post, we will look at the basic syntax of this cool feature, its use cases, and even some real-life code examples.

9.1 The basics of structured binding

Starting from C++17, you can write:

```
1 std::set<int> mySet;
2 auto [iter, inserted] = mySet.insert(10);
```

`insert()` returns `std::pair` indicating if the element was inserted or not, and the iterator to this element. Instead of `pair.first` and `pair.second`, you can use variables with concrete names.

(You can also assign the result to your variables by using `std::tie()`; still, this technique is not as convenient as structured bindings in C++17)

Such syntax is called a structured binding expression.

9.2 The Syntax

The basic syntax for structured bindings is as follows:

```
1 auto [a, b, c, ...] = expression;
2 auto [a, b, c, ...] { expression };
3 auto [a, b, c, ...] ( expression );
```

The compiler introduces all identifiers from the `a, b, c, ...` list as names in the surrounding scope and binds them to sub-objects or elements of the object denoted by `expression`.

Behind the scenes, the compiler might generate the following **pseudo code**:

```

1 auto tempTuple = expression;
2 using a = tempTuple.first;
3 using b = tempTuple.second;
4 using c = tempTuple.third;

```

Conceptually, the expression is copied into a tuple-like object (`tempTuple`) with member variables that are exposed through `a`, `b` and `c`. However, the variables `a`, `b`, and `c` are not references; they are aliases (or bindings) to the generated object member variables. The temporary object has a unique name assigned by the compiler.

For example:

```

1 std::pair a(0, 1.0f);
2 auto [x, y] = a;

```

9.3 Modifiers

Several modifiers can be used with structured bindings (similarly like on `auto`):

`const` modifiers:

```
1 const auto [a, b, c, ...] = expression;
```

References:

```

1 auto& [a, b, c, ...] = expression;
2 auto&& [a, b, c, ...] = expression;

```

For example:

```

1 std::pair a(0, 1.0f);
2 auto& [x, y] = a;
3 x = 10; // write access
4 // a.first is now 10

```

In the example, `x` binds to the element in the generated object, which is a reference to `a`.

Now it's also relatively easy to get a reference to a tuple member:

```
1 auto& [refA, refB, refC, refD] = myTuple;
```

Or better via a `const` reference:

```
1 const auto& [refA, refB, refC, refD] = myTuple;
```

You can also add `[[attribute]]` to structured bindings:

```
1 [[maybe_unused]] auto& [a, b, c, ...] = expression;
```

9.4 Binding

Structured Binding is not only limited to tuples; we have three cases from which we can bind from:

1. If the initializer is an array:

```

1 // works with arrays:
2 double myArray[3] = { 1.0, 2.0, 3.0 };
3 auto [a, b, c] = myArray;

```

In this case, an array is copied into a temporary object, and a, b, and c refers to copied elements from the array.

The number of identifiers must match the number of elements in the array.

2. If the initializer supports `std::tuple_size`, provides `get<N>()` and also exposes `std::tuple_element` functions:

```

1 std::pair myPair(0, 1.0f);
2 auto [a, b] = myPair; // binds myPair.first/second

```

In the above snippet, we bind to `myPair`. But this also means you can provide support for your classes, assuming you add the `get<N>` interface implementation. See an example in the later section.

3. If the initializer's type contains only non-static data members:

```

1 struct Point {
2     double x;
3     double y;
4 };
5
6 Point GetStartPoint() {
7     return { 0.0, 0.0 };
8 }
9
10 const auto [x, y] = GetStartPoint();

```

x and y refer to `Point::x` and `Point::y` from the `Point` structure.

The class doesn't have to be `POD`, but the number of identifiers must equal to the number of non-static data members. The members must also be accessible from the given context.

9.5 C++17/C++20 changes

During the work on C++20, there were several proposals that improved the initial support for structured bindings:

- P0961 - Relaxing the structured bindings customization point finding rules
- P0969 - Allow structured bindings to accessible members
- P1091 - Extending structured bindings to be more like variable declarations
- P1381 - Reference capture of structured bindings

It looks like GCC implemented those features working even for the C++17 mode.

For example, you can even capture them in lambda:

```

1 std::pair xy { 42.3, 100.1 };
2 auto [x, y] = xy;

```

```

3 auto foo = [&x, &y]() {
4     std::cout << std::format("{}", {}, x, y);
5 };
6 foo();

```

9.6 Iterating through maps

If you have a `std::map` of elements, you might know that internally, they are stored as pairs of `<const Key, ValueType>`.

Now, when you iterate through elements of that map:

```

1 for (const auto& elem : myMap) { ... }

```

You need to write `elem.first` and `elem.second` to refer to the key and value. One of the coolest use cases of structured binding is that we can use it inside a range based for loop:

```

1 std::map<KeyType, ValueType> myMap;
2 // C++14:
3 for (const auto& elem : myMap) {
4     // elem.first - is value key
5     // elem.second - is the value
6 }
7 // C++17:
8 for (const auto& [key, val] : myMap) {
9     // use key/value directly
10 }

```

In the above example, we bind to a pair of `[key, val]` so we can use those names in the loop. Before C++17, you had to operate on an iterator from the map - which returns a pair `<first, second>`. Using the real names `key/value` is more expressive.

The above technique can be used in the following example:

```

1 #include <map>
2 #include <iostream>
3
4 int main() {
5     const std::map<std::string, int> mapCityPopulation {
6         { "Beijing", 21'707'000 },
7         { "London", 8'787'892 },
8         { "New York", 8'622'698 }
9     };
10
11     for (const auto&[city, population] : mapCityPopulation)
12         std::cout << city << ":" << population << '\n';
13 }

```

In the loop body, you can safely use the `city` and `population` variables.

9.7 Working with structures and arrays

Here's another example where structured binding might be handy:

```

1 #include <iostream>
2 #include <format>
3 #include <array>
4
5 struct Color {
6     // internal representation...
7     int data { 0 };
8     int model { 0 };
9
10    std::array<unsigned char, 3> getRGB() const {
11        // do some conversion from the internal model...
12        return {255, 128, 255 };
13    }
14    std::array<double, 3> getHSV() const {
15        // do some conversion from the internal model...
16        return {0.5, 0.1, 1. };
17    }
18};
19
20 int main() {
21     Color col{};
22     auto [r, g, b] = col.getRGB();
23     std::cout << std::format("{} {}, {}\n", r, g, b);
24     auto [h, s, v] = col.getHSV();
25     std::cout << std::format("{} {}, {}\n", h, s, v);
26 }
```

As you can see, the code has a `Color` structure that might contain some internal representation, and you use getters to get RGB or HSV models. Thanks to the structured binding, it's easy to name those sub-objects, rather than rely on arrays.

9.8 Real code

Let's have a look at some code in the wild. I found the following use cases:

In Tensorflow: `nccl_collective_thunk.cc`¹

```

1 auto [_, was_inserted] =
2 done_events_.insert({device_ordinal, std::move(done_event)});
```

Powertoys Microsoft: `updating.cpp`²

¹https://github.com/tensorflow/tensorflow/blob/82c38106d03c15efe3ee53401e3527348a62a8cb/tensorflow/compiler/xla/service/gpu/nccl_collective_thunk.cc#L222

²<https://github.com/microsoft/PowerToys/blob/c2325181cae1efa3a4786752f323ab1c6686965b/src/common/updating/updating.cpp#L122>

```

1 auto [installer_download_url, installer_filename]
2     = extract_installer_asset_download_info(release_object);
3 co_return new_version_download_info{ extract_release_page_url(release_object),
4                                         std::move(github_version),
5                                         std::move(installer_download_url),
6                                         std::move(installer_filename) };

```

Microsoft Terminal: state.cpp³

```

1 const auto [colorForeground, colorBackground]
2     = renderSettings.GetAttributeColors(textAttributes);

std::from_chars() and bitcoin::bitcoin4

1 const auto [first_nonmatching, error_condition]
2     = std::from_chars(val.data(), val.data() + val.size(), result);

```

³<https://github.com/microsoft/terminal/blob/772ed3a7b7c41bc8c04feb6e6ae1638343dcbf15/src/renderer/gdi/state.cpp#L277>

⁴<https://github.com/bitcoin/bitcoin/blob/7386da7a0b08cd2df8ba88dae1fab9d36424b15c/src/univalue/include/univalue.h#L143>

Chapter 10

std::optional and non-POD C++ types

👤 Felipe 📅 2021-09-19 🏷️ ★★★

C++ 17 has introduced the `std::optional` template class¹ that is analogous to the Maybe/Optional monad² implementation in many other languages. “Analogous” is doing a lot of work in this statement because the C++ type checker is not going to help you³ avoid dereferencing an empty optional like Rust, Haskell, Scala, Kotlin, TypeScript and many other languages will do.

That does not make it useless. As with many things in C++, we will be careful™ when using it and write only programs that do not dereference an empty optional.

In languages that deal mostly with reference types⁴, an optional type can be implemented as an object that wraps a reference and a tag bit that tells if the optional has some data or nothing.⁵ In C++ on the other hand, the `std::optional` will inline the value type `T` onto itself. That means the general behavior of an optional of `T` depends a lot on the specifics of the type it’s wrapping.

For integer, floats, characters, the use of `std::optional` doesn’t bring many surprises. In this post I want to look at what happens when non-POD⁶ types are wrapped in an optional. For this, I will write a class that prints a different message on each special function call:

```
1  class Object {
2  private:
3     std::string _s;
4
5  public:
6     Object() { puts("default-constructed"); }
7     ~Object() { puts("destroyed"); }
8
9     explicit Object(const std::string &s) : _s(s) { puts("constructed"); }
10
11    Object(const Object &m) : _s(m._s) { puts("copy-constructed"); }
```

¹<https://en.cppreference.com/w/cpp/utility/optional>

²<https://wiki.haskell.org/Maybe>

³https://en.cppreference.com/w/cpp/utility/optional/operator_star

⁴https://en.wikipedia.org/wiki/Value_type_and_reference_type

⁵<https://felipe.rs/2021/09/19/std-optional-and-non-pod-types-in-cpp/#fn:flowtyping>

⁶https://en.wikipedia.org/wiki/Passive_data_structure

```

12     Object &operator=(const Object &m) {
13         puts("copy-assinged");
14         _s = m._s;
15         return *this;
16     }
17
18     Object(Object &&m) : _s(std::move(m._s)) { puts("move-constructed"); }
19     Object &operator=(Object &&m) {
20         puts("move-assigned");
21         _s = std::move(m._s);
22         return *this;
23     }
24
25     void dump() const { puts(_s.c_str()); }
26 };

```

And write a function that returns an optional of this class — a common use-case of optionals. The returned optional will contain a value if the string argument is non-empty, and be `std::nullopt` otherwise.

```

1 std::optional<Object> maybe(const std::string &s) {
2     if (s.empty()) {
3         return std::nullopt;
4     }
5     return Object(s);
6 }

```

10.1 The good

When using a `std::optional`, neither the `Object` constructors or destructors have to be called if the variable never gets populated with a value.

To see this in action, consider `program1`

```

1 void program1(const std::string &s) {
2     const std::optional<Object> o = maybe(s);
3     if (o) {
4         o->dump();
5     } else {
6         puts("<empty>");
7     }
8 }

```

and its output when called with an empty string

```

1 <empty>

```

10.2 The ugly

Things get more involved when `program1` is called with “Hello!” and the optional gets populated

```

1 constructed
2 move-constructed
3 destroyed
4 Hello!
5 destroyed

```

The `return Object(s)` line in `maybe`, calls `Object::Object(const std::string &)` to create the `Object` that then gets moved into the storage within the `std::optional`. If `Object` didn’t have a move-constructor, it would be copied here. At the end of the scope of `maybe`, the “moved-from” temporary `Object` instance is destroyed, and at the caller —`program1`— `Object::~Object` has to be implicitly called again to destroy the `Object` within the `std::optional` instance.

This situation can be improved if we tell `std::optional<Object>` to forward⁷ the arguments to `Object::Object` so it can construct `Object` in the optional’s storage area right away without a temporary

```

1 if (s.empty()) {
2     return std::nullopt;
3 }
4 - return return Object(s);
5 + return std::optional<Object>(s);

```

With this change, the output of `program1("Hello!")` becomes

```

1 constructed
2 Hello!
3 destroyed

```

Only one constructor invocation and one destructor invocation. A win!

However, most functions returning a `std::optional<T>` are calling some function that returns `T` in the code path that instantiates and returns the optional. That takes us back to the same situation of duplicated constructor/destructor invocations.

```

1 Object makeObject(const std::string &s);
2
3 std::optional<Object> maybe(const std::string &s) {
4     if (s.empty()) {
5         return std::nullopt;
6     }
7     return makeObject(s);
8 }

```

To improve this and keep the logic of `makeObject` separate from `maybe`, we would have to change `makeObject` to allow the perfect-forwarding⁸ of the parameters from `maybe`, to `makeObject`, to `std::optional::optional`, to `Object::Object`!

⁷<https://eli.thegreenplace.net/2014/perfect-forwarding-and-universal-references-in-c/>

⁸<https://eli.thegreenplace.net/2014/perfect-forwarding-and-universal-references-in-c/>

Another common way of writing these functions is by declaring a variable of type `T`, performing some operations on it, and then returning it wrapped in an optional. This has the same problem we started with.

```

1 std::optional<Object> maybe(const std::string &s) {
2     if (s.empty()) {
3         return std::nullopt;
4     }
5     Object o(s);
6     doSomething(o);
7     return o;
8 }
```

10.3 The bad

These problems might not be a big deal in most situations, but if you insist on returning non-PODs wrapped in a optional, make sure that:

- The wrapped type should be cheaply movable, otherwise your program might be copying it on every function call due to innocent-looking code;
- Define your destructors outside the class declaration so they don't get inlined by the compiler in both functions — the caller and the callee that returns the optional — to avoid binary size increase.

The specific situation in which I've seen this really affect binary size and possibly performance is when functions are written to return instances of classes generated by the Google Protocol Buffers ⁹compiler.

Google Protocol Buffers for C++ was designed before C++11 (i.e. before move-semantics was added to the language). Its APIs and generated code are designed to make it possible to use the classes without ever invoking copy constructors. It's a good API.

If you never invoke a function, it doesn't need to be in the compiled binary. You can notice a sudden increase in the binary size of your program when a single call to a big function is added to the codebase. Returning an optional of a Protocol Buffers object is enough to instantiate a lot of code that could otherwise never be needed.

Let's take a look at the generated code based on a Protocol Buffers message called `Date`

```
class Date PROTOBUF_FINAL : public ::PROTOBUF_NAMESPACE_ID::Message { public: inline
```

```
Date() : Date(nullptr) {} virtual ~Date();
```

```

1 class Date PROTOBUF_FINAL : public ::PROTOBUF_NAMESPACE_ID::Message {
2 public:
3     inline Date() : Date(nullptr) {}
4     virtual ~Date();
5
6     Date(const Date& from);
7     Date(Date&& from) noexcept
8         : Date() {
```

⁹<https://protobuf.dev/reference/cpp/cpp-generated/>

```

9     *this = ::std::move(from);
10    }
11
12    inline Date& operator=(const Date& from) {
13        CopyFrom(from);
14        return *this;
15    }
16    inline Date& operator=(Date&& from) noexcept {
17        if (GetArena() == from.GetArena()) {
18            if (this != &from) InternalSwap(&from);
19        } else {
20            CopyFrom(from);
21        }
22        return *this;
23    }
24    ...

```

The move-constructor (by calling `operator=(Date &&)`) can potentially call `InternalSwap` and `CopyFrom`. The latter is called when the objects can't be swapped because they are allocated in different arenas and have to be copied instead. By using the move-constructor of this object, both the moving (swapping) and copying functions are instantiated in the binary. This explains why returning the optional of a Protocol Buffers class increases the binary size of a program that, before doing that, didn't have a need for `InternalSwap` and the `CopyFrom` function.

10.4 Recommendation

By adopting a more C-like way of initializing structures, the unnecessary use of move-constructors and extraneous destructor calls can be avoided. This pattern fits nicely with the code generated by Protocol Buffers.

Let's add a new member function to the `Object` class

```
1 void set(const std::string &s) { _s = s; }
```

and write an alternative to `program1` — `program2`

```

1 [[nodiscard]] bool maybe(const std::string &s, Object &out_object) {
2     if (s.empty()) {
3         return false;
4     }
5     out_object.set(s);
6     return true;
7 }
8
9 void program2(const std::string &s) {
10     Object o;

```

```

11     if (maybe(s, o)) {
12         o.dump();
13     } else {
14         puts("<empty>");
15     }
16 }
```

The `maybe` function was rewritten to take an output parameter and return a boolean. `out_object` is changed in-place and doesn't have to be moved into an optional and then destroyed within `maybe`. As expected, `program2("Hello!")` generates a cleaner output

```

1 default-constructed
2 Hello!
3 destroyed
```

`program2` does not have to ever call the move-constructor, so it can be discarded by the linker and the destructor is called only once. If the destructor was inlined, it would be inlined once in the program, not twice.

10.5 Conclusion

Optionals are far from zero-cost abstractions in C++ and if this cost matters to you, taking output parameters and returning non-discardable booleans is an advantageous alternative solution to a function returning `std::optional<T>` when objects of type `T` are expensive¹⁰ to move and/or destroy.

1. Languages like TypeScript can statically determine if an object is set based on its position in the control flow of the program.¹¹
2. In the sense of run-time and size of the code in the binary. □

¹⁰<https://felipe.rs/2021/09/19/std-optional-and-non-pod-types-in-cpp/#fn:expensive>

¹¹https://en.wikipedia.org/wiki/Flowsensitive_typing

Chapter 11

C++20 concepts are structural: What, why, and how to change it?

👤 Jonathan 📅 2021-07-29 💬 ★★★★

C++20 added concepts as a language feature. They're¹ often² compared³ to Haskell's type classes⁴, Rust's traits⁵ or Swift's protocols⁶.

Yet there is one feature that sets them apart: types model C++ concepts automatically. In Haskell, you need an `instance`, in Rust, you need an `impl`, and in Swift, you need an `extension`. But in C++? In C++, concepts are just fancy boolean predicates that check for well-formed syntax: every type that makes the syntax well-formed passes the predicate and thus models the concepts.

This was the correct choice, but is sometimes not what you want. Let's explore it further.

11.1 Nominal vs. structural concepts

To co-opt terms from type systems, C++20 concepts use structural typing⁷: a type models the concept if it has the same structure as the one required by the concept, i.e. it has required expressions. On the contrast, type classes, traits and protocols all use nominal typing: a type models the concept only if the user has written a declaration to indicate it.

For example, consider a C++ concept that checks for `operator==` and `operator!=`:

```
1 template <typename T>
2 concept equality_comparable = requires (T obj) {
3     { obj == obj } -> std::same_as<bool>;
4     { obj != obj } -> std::same_as<bool>;
5 };
```

¹<https://stackoverflow.com/questions/32124627/how-are-c-concepts-different-to-haskell-typeclasses>

²<https://stackoverflow.com/questions/56045846/what-are-the-similarities-and-differences-between-cs-concepts-and-rusts-trai>

³<https://www.youtube.com/watch?v=Qh7QdG5RK9E>

⁴https://en.wikibooks.org/wiki/Haskell/Classes_and_types

⁵<https://doc.rust-lang.org/book/ch10-02-traits.html>

⁶<https://docs.swift.org/swift-book/LanguageGuide/Protocols.html>

⁷https://en.wikipedia.org/wiki/Structural_type_system

This is how you write a type that models `equality_comparable` with C++20's structural concepts:

```

1 // Define your type,
2 struct vec2
3 {
4     float x, y;
5
6     // define the required operators,
7     friend bool operator==(vec2 lhs, vec2 rhs)
8     {
9         return lhs.x == rhs.x && lhs.y == rhs.y;
10    }
11
12     // operator!= not needed in C++20 due to operator rewrite rules!
13 };
14
15 // ... and that's it!
16 static_assert(equality_comparable<vec2>);

```

In contrast, this is how you would write a type that models `equality_comparable` in a hypothetical C++20 with nominal concepts:

```

1 // Define your type
2 struct vec2 { ... }; // as before
3
4 // ... and tell the compiler that it should be `equality_comparable`.
5 // Most languages also support a way to define the operation here.
6 concept equality_comparable for vec2;

```

11.2 Nominal is better...

In my opinion, nominal concepts are superior to structural concepts:

1. Structural concepts do not allow for semantic differences between concepts, because that is not part of the “structure”.

Consider the standard library concept `std::relation`; it is true for predicate types `R` that describe a binary relation between types `T` and `U`:

```

1 template <typename F, typename ... Args>
2 concept predicate
3     = /* F can be invoked with Args returning bool */;
4
5 template <typename R, typename T, typename U>
6 concept relation = predicate<R, T, T> && predicate<R, U, U>
7                 && predicate<R, T, U> && predicate<R, U, T>;

```

Binary relations⁸ are broad mathematical terms, so often you want a relation with specific properties. For example, `std::ranges::sort` takes a function that controls the sort, which must be a special relation: a strict weak order⁹. Luckily, there is a standard library concept `std::strict_weak_order`:

```
1 template <typename R, typename T, typename U>
2 concept strict_weak_order = relation<R, T, U>;
```

However, it is just `std::relation`! Whether you use `requires std::strict_weak_order` or `requires std::relation` makes as much difference as calling your template parameters `RandomAccessIterator`. It's just a fancy comment; the compiler doesn't care.

Semantic differences that cannot be expressed in the C++ type system can't be expressed with structural concepts either. With nominal concepts, a function object would need to explicitly opt-in to `strict_weak_order`, which allows differentiating between the two.

- With structural concepts, names of functions are really important (ironic, I know). If you write code that interacts with the standard library (or other libraries using concepts) in any way, you need to follow the same naming convention. Names like `size` or `begin` or `iterator` are essentially globally reserved and must mean the thing the standard library concepts intend.

```
1 class TShirt
2 {
3     public:
4         enum Size
5         {
6             small,
7             medium,
8             large
9         };
10
11     // The size of the T-Shirt.
12     Size size() const;
13
14     // The text on the front of the T-Shirt.
15     const std::string& front() const;
16     // The text on the back of the T-Shirt.
17     const std::string& back() const;
18 }
```

The `TShirt` class above might be mistaken for some sequence container like `std::vector` as it passes the syntactic checks of corresponding concepts. However, with nominal concepts it would need to explicitly opt-in; no type will model a nominal concept if the author did not intend it.

- On the flip side, if we have something that conceptually models a concept, but uses different names for the required methods, it does not work – as the name is what matters.

⁸https://en.wikipedia.org/wiki/Binary_relation

⁹<https://www.foonathan.net/2018/07/ordering-relations-math/>

Suppose `vec2` from above didn't overload `operator==` but instead provided a function `bool is_equal()`:

```

1   struct vec2
2   {
3       float x, y;
4
5       bool is_equal(vec2 rhs) const
6       {
7           return x == rhs.x && y == rhs.y;
8       }
9   };

```

Even though the type is equality comparable, it is not `equality_comparable` – names matter. With nominal concepts, the declaration that opts-in to a concept usually also provides a way to specify the actual implementation of the required functions. That way, you can easily adapt existing types to other interfaces:

```

1 // Dear compiler, vec2 models equality_comparable and here's how:
2 concept equality_comparable for vec2
3 {
4     bool operator==(vec2 lhs, vec2 rhs)
5     {
6         return lhs.is_equal(rhs);
7     }
8 }

```

One can imagine that the names introduced there are scoped to the concept: They don't add members to the type itself and are instead only available in generic code that wants `equality_comparable` types.

11.3 but structural is what C++ needs

So if I believe that nominal concepts are better, why did I say in the introduction that structural concepts were the correct choice for C++? Because structural concepts have one big advantage: they're convenient when faced with code written before concepts!

Just imagine if every function conceptified in C++20 requires you to explicitly opt-in to the concepts: you can't use `std::ranges::sort()` until you've written dummy declarations for your containers, your iterators, your types. It would be a migration nightmare! It's much easier if the concept is modeled automatically.

Another advantage is library interoperability: if you have three libraries A, B, and C, where A has a concept, B has a type that models the concept, and C uses the two, C can just pass the type of B to functions expecting A's concept without B having to depend on A or C. You can write types that adhere to concepts without pulling in the library that actually defines them, which is convenient when you want to avoid a big dependency yet still allow your code to seamlessly work with it.

This argument is sometimes phrased slightly differently: C can pass B's type to A's concept without B ever knowing about the existence of A's concept. However, this does not happen: you

don't write a type that just so happens to model a concept that you're unaware of. What does happen is that you write a type that models an existing concept, which just hasn't been formalized in code yet.

Finally, sometimes a naming convention is just so universally accepted that nobody would ever dare and deviate from it – think operators. If you're copy assignment doesn't do a copy, or your move constructor doesn't move, your type is bad. It thus makes total sense to have concepts like `std::copyable` be modeled automatically.

Note that all three advantages don't apply to “new” languages, i.e. where concepts are part of it from the start:

- A new language has no legacy code, so there is no migration cost to annotating every concept your type models.
- A new language can provide a standard package manager, which makes it less necessary to avoid dependencies to model concepts.
- Instead of having operator overloading and concepts that check for their existence, you can flip it on its head: Define a concept that provides the operator overloads; type that opt-in to the concept get the corresponding overloaded operator.

As such, the decision of Haskell, Rust, and Swift makes perfect sense.

However, when you invent completely novel concepts for a library or actually need to distinguish between different concepts based on semantics – and don't just want “fancy comments”, you might want nominal concepts in C++.

So what do you do?

11.4 Nominal concepts in C++20

The problem of differentiating between concepts with identical interface but different semantics dates back to C++98 – iterators. An input iterator and a forward iterator have (almost?) the same interface, yet are not interchangeable: once you advance an input iterator it is gone and you'll never get the old value back; with a forward iterator, you can copy it and retain the old value.

```

1 template <typename InputIterator>
2 void handle_input(InputIterator begin, InputIterator end)
3 {
4     ...
5
6     auto a = *begin;
7
8     auto copy = begin;
9     ++begin;
10    auto b = *begin;
11
12    ...

```

```

13
14     auto c = *copy;
15     assert(c == a); // ups, c is actually the same value as b!
16 }
```

Because copying an input iterator doesn't really make sense as all input iterators share the same state, in C++20 the conceptified `std::input_iterator` does not require the existence of a copy constructor, and many input iterators of the standard library are move-only types.

This is one of my two favorite features of the new iterator model, besides sentinels¹⁰.

So how can code distinguish between an input iterator and a forward iterator? Simple: we add some syntax that distinguishes them.

In the case of iterators, every iterator has an associated `iterator_category` typedef that explicitly states whether something is an input iterator (`std::input_iterator_tag`) or a forward iterator iterator (`std::forward_iterator_tag`). In fact, there are iterator categories for all iterator categories as C++98 wasn't really great for detecting the interface of a type and doing overloading based on that...

However, the basic idea to distinguish semantic properties using tag types was kept for the new C++20 iterator concepts. The required typedef is now called `iterator_concept` for reasons, but it also looks for `iterator_tag`.

Technique #1: add extra syntax like a dummy typedef that distinguishes between otherwise identical concepts.

```

1 // concept definition ===/
2 template <typename T>
3 concept my_concept
4     = requires { typename T::my_concept_tag; }
5     && ...;
6
7 //==== concept modelling ===/
8 struct my_type_modelling_the_concept
9 {
10     using my_concept_tag = void; // Doesn't matter.
11 };
```

Another case is the distinction between `std::range` and `std::view`. A `std::view` is a `std::range` (something with begin/end) that is also moveable, but where move and copy operations (if provided) happen in constant time. So crucially, `std::vector` is not a `std::view`: it has begin/end, is moveable (and even copyable) but copy operations are certainly not in O(1)! As such, `std::vector` is not a `std::view` – which is again impossible to detect by a compiler because it has the same syntax.

So to model a `std::view` a type has to opt-in by specializing the variable template `std::enable_view` to set it to `true`:

¹⁰<https://www.foonathan.net/2020/03/iterator-sentinel/>

```

1  namespace my_namespace
2  {
3      class MyViewtype
4      {
5          public:
6              iterator begin() const;
7              iterator end() const;
8      };
9  }
10
11 namespace std
12 {
13     // Tell the compiler that your view is a view.
14     template <>
15     constexpr bool enable_view<my_namespace::MyViewType> = true;
16 }
```

If you compare this with the `equality_comparable` nominal concept example from above, you'll note that it basically looks the same! We formally fulfill the syntactic requirements for our type, and then write some extra declaration to indicate that we'd like to model the concept. It's just purely implemented in the library, instead of the core language.

However, specialization of `std` things is annoying (close the current namespace, open namespace `std`, write a `template`, so there is also an easier way to opt-in: you simply inherit from `std::view_base`.

```

1  namespace my_namespace
2  {
3      // Tell the compiler that your view is a view.
4      class MyViewtype : public std::view_base
5      {
6          public:
7              iterator begin() const;
8              iterator end() const;
9      };
10 }
```

This is not inheritance with virtual functions or CRTP (although there is also a CRTP base class for views¹¹) or anything like that: `std::view_base` is simply an empty type. Its only there to be able to provide a syntactic requirement that can be checked by the non-specialized version of `std::enable_view`:

```

1  namespace std
2  {
3      struct view_base
4      {};
5  }
```

¹¹https://en.cppreference.com/w/cpp/ranges/view_interface

```

6   // By default, a type is a view iff it inherits from view_base.
7   template <typename T>
8   constexpr bool enable_view = std::is_base_of_v<view_base, T>;
9 }
```

Technique #2: enable a concept by specializing a variable template and/or inheriting from a tag type

```

1 //==== concept definition ====
2 struct my_concept_base {};
3
4 template <typename T>
5 constexpr bool enable_my_concept
6 = std::is_base_of_v<my_concept_base, T>;
7
8 template <typename T>
9 concept my_concept = enable_my_concept<T>
10 && requires (T obj) { ... };
11
12 //==== concept modelling ====
13 struct my_type_modelling_the_concept : my_concept_base
14 {
15 ...
16 };
```

The extra layer of indirection added by the variable template is only necessary, if some types want to model `my_concept` but can't inherit from `my_concept_base` (non-class types, preexisting types). If you're adding a completely new concept that is only ever modeled by classes, you can just use `std::is_base_of_v` directly.

I really like the “enable a concept by inheriting from a tag type” idiom (EACBIFATT?): it provides nominal concepts with minimal syntactic overhead to opt-in. We can also extend the base class to inject default implementations for optional functionality, which can be “overridden” by simple name hiding.

The technique also has the “advantage” of adding your library’s namespace to the list of namespaces found via ADL of your users types!

For `lexy`¹², where rules inherit from `lexy::dsl::rule_base`, this means it finds the DSL operator overloads automatically, so that worked out nicely. For all other libraries, it is probably not what you want.

Now you might wonder: if users need to explicitly inherit something anyways, why not use that alone to constrain the function? After all, it worked for iterators since C++98.

However, consider the case where a type claims to model a concept, but actually doesn’t. With the additional syntax checks, you’ll get an error message when trying to call the function. Without concepts, it is somewhere in the internals when the code tries to use the type.

¹²<https://github.com/foonathan/lexy>

Whether or not that is worth it, is up to you. For example, lexy, which supports C++17, can only use concepts by hiding them behind ugly macros. As such, I didn't bother to properly conceptify my concepts and only use the existence of base classes.

11.5 Reverse nominal concepts

On the flip side, sometimes you don't want to explicitly opt-in to a concept, but to opt-out.

For example, a `std::sized_range` is a `std::range` with a `size()` function that returns the size in constant time. Again, this cannot be verified by the compiler, so there needs an additional nominal check. We can again throw EACBIFATT on it, but this would be annoying: most `size()` functions are O(1).

So instead the logic is reversed: by default types model the concept if they fulfill the syntactic requirements, unless you've opted-out by specializing `disable_sized_range`.

```

1  namespace std
2  {
3      // MyLinkedList has O(n) size.
4      template <typename T>
5      constexpr bool disable_sized_range<MyLinkedList<T>> = true;
6  }
```

Technique #3: explicitly disable a concept by specializing a variable template

```

1  template <typename T>
2  constexpr bool disable_my_concept = false;
3
4  template <typename T>
5  concept my_concept = !disable_my_concept<T>
6  && requires (T obj) { ... };
```

Note that we could again provide the tag type to inherit, but inheriting something to opt-out seems weird.

11.6 Conclusion

C++20 concepts are automatically modeled based on syntax; it doesn't care about semantics.

As such, if you want to distinguish between identical syntax with different semantics, you need to introduce some syntax to distinguish it. A nice way is to check for the existence of a base class: types can easily opt-in by inheriting from it. You can also add typedefs or variable specializations. The same approach can also be used to opt-out of a concept.

Chapter 12

Most C++ constructors should be explicit

• Arthur O' Dwyer 📅 2023-04-08 🔮 ★★★

All your constructors should be explicit by default. Non-explicit constructors are for special cases.

The `explicit` keyword disallows “implicit conversion” from single arguments or braced initializers. Whereas a non-`explicit` constructor enables implicit conversion —

```
1 struct Im {  
2     Im();  
3     Im(int);  
4     Im(int, int);  
5 };  
6 void read_im(const Im&);  
7 void test_im() {  
8     Im i1;  
9     Im i2 = Im();  
10    Im i3 = Im(1);  
11    Im i4 = Im(1, 2);  
12    Im i5 = {};  
13    Im i6 = 1;  
14    Im i7 = {1};  
15    Im i8 = {1, 2};  
16    read_im({});  
17    read_im(1);  
18    read_im({1});  
19    read_im({1, 2});  
20 }
```

—an `explicit` constructor strictly limits the number of syntaxes by which that constructor can be

invoked —

```

1  struct Ex {
2      explicit Ex();
3      explicit Ex(int);
4      explicit Ex(int, int);
5  };
6  void read_ex(const Ex&);
7
8  void test_ex() {
9      Ex e1;
10     Ex e2 = Ex();
11     Ex e3 = Ex(1);
12     Ex e4 = Ex(1, 2);
13     read_ex(Ex());
14     read_ex(Ex(1));
15     read_ex(Ex(1, 2));
16 }
```

I claim that the latter is *almost always* what you want, in production code that needs to be read and modified by more than one person. In short, **explicit is better than implicit**.

12.1 C++ gets the defaults wrong

C++ famously “gets all the defaults wrong” :

- `switch` cases fall through by default; you have to write `break` by hand.
- Local variables are uninitialized by default; you must write `=0` by hand. (In a just world, there’d be loud syntax for “this variable is uninitialized,” and quiet syntax for “this variable is value-initialized to zero.”)
- Most variables won’t¹; but C++ makes non-const the default, so that you must write `const` by hand in many places. (But please not too many!²)
- Most classes aren’t actually intended as bases for inheritance, but C++ permits deriving from any class, unless you write `final` by hand.
- Constructors correspond to implicit conversions by default; to get them to behave only like the explicit constructors in any other language, you must write `explicit` by hand.
- Almost all the constructors in the STL are implicit rather than `explicit`; for example, `{&i, &j}` implicitly converts to `vector<int>`, which means so does `{"a", "b"}3`.

¹<https://www.hpcalc.org/hp48/docs/columns/aphorism.html>

²<https://quuxplusone.github.io/blog/2022/01/23/dont-const-all-the-things/>

³<https://godbolt.org/z/TWYrqoc4x>

Most of these defaults merely grant license to do things that sane code doesn’t do anyway; arguably the wrong defaults can be ignored. Personally I don’t recommend writing `const` on all the things⁴, nor writing `final` on non-polymorphic classes. But I do recommend explicitly initializing scalar variables and data members, and of course you have to write `break` in the right places. I treat the `explicit` keyword as somewhere in that latter category: writing `explicit` on all your constructors is at least as important as explicitly initializing all your scalar variables.

I say you should write `explicit` on “all” your constructors. What I really mean is, you should write it on 99% of your constructors. There are a handful of special cases —literally, I can think of four—where it’s correct to leave a constructor as non-explicit.

12.2 Implicit is correct for copy and move constructors

C++ loves to make implicit copies of things. If you marked your copy constructor as `explicit`, then simple copying wouldn’t work anymore:

```
A a1;
A a2 = a1;
// no matching constructor for initialization of `a2`
```

So never mark a single-argument copy or move constructor as `explicit`. But do continue to mark your zero-argument constructor `explicit`; there’s no problem with the initialization of `a1` here.

12.3 Implicit is correct for types that behave like bags of data members

In C, the notion of “`struct` type” or “array type” is essentially identical with “these elements, in this order.” So in C, we always initialize structs and arrays with curly braces because this kind of type —the aggregate—is all we have to work with.

```
1 struct CBook {
2     const char *title;
3     const char *author;
4 };
5 struct CBook mycbook = { "Hamlet", "Shakespeare" };
```

C++ not only adopted the notion of aggregate types directly from C (for backward compatibility), but also modeled its class types a little too much on C’s aggregates. The archetypical C++ class is a “bag of data members” :

```
1 class Book {
2     std::string title_;
3     std::string author_;
4 public:
5     Book(std::string t, std::string a) :
6         title_(std::move(t)), author_(std::move(a)) {}
```

⁴<https://quuxplusone.github.io/blog/2022/01/23/dont-const-all-the-things/>

```

7     std::string title() const { return title_; }
8     std::string author() const { return author_; }
9 };
10
11 int add_to_library(const Book&);
12
13 Book mybook = { "Hamlet", "Shakespeare" };

```

If our intent is that a Book should be identical with the notion of “a title plus an author, in that order,” forever, then there is absolutely nothing wrong with treating `{"Hamlet", "Shakespeare"}` as a Book. That’s just “uniform initialization,” the same thing `std::pair<std::string, std::string>` does.

But in the real world, “bags of data members” are surprisingly uncommon. If you think you’ve found one, you’ll probably be surprised in a few years to find out that you were wrong! For example, we might eventually realize that every Book also has a pagecount. So we change our class to

```

1 class Book {
2     std::string title_;
3     std::string author_;
4     int pagecount_;
5 public:
6     Book(std::string t, std::string a, int p) :
7         title_(std::move(t)), author_(std::move(a)), pagecount_(p) {}
8     // ~~~
9 };

```

Now `Book("Hamlet", "Shakespeare")` is no longer a valid expression; we are forced to go find all the places that explicitly construct Books and update them. However, the braced initializer `{"Hamlet", "Shakespeare"}` remains valid; it’s just no longer implicitly convertible to Book. It might now prefer to convert to something else. For example, consider this overload set (<https://godbolt.org/z/d8T4f6dGj>):

```

1 int add_to_library(const Book&); // #1
2
3 template<class T = void>
4 int add_to_library(std::pair<bool, const T*>); // #2
5
6 add_to_library({"Hamlet", "Shakespeare"});

```

If Book’s implicit constructor takes two `std::strings`, then this is a call to `add_to_library(const Book&)` with a temporary Book. But if Book’s implicit constructor takes two strings and an `int`, then this is a call to `add_to_library(pair<bool, const void*>)`. There’s no ambiguity in either case as far as the compiler is concerned. Any ambiguity you and I see in this code is ambiguity in the human sense, inserted by the human programmer who decided that `{"Hamlet", "Shakespeare"}` was all it took—and all it would take, forever—to make a Book.

Software engineering is programming integrated over time.

— “Software Engineering at Google” (Titus Winters, Tom Manschreck, Hyrum Wright)

So if our Book is not just a bag of data members, like `std::pair` or `std::tuple` is, then this particular excuse fails to apply to our Book. There's a good way to tell if this excuse applies: Is your type actually named `std::pair` or `std::tuple`? No? Then it's probably not a bag of data members.

```
pair(first_type f, second_type s);
// non-explicit
```

On the flip side: If you're considering making a type that is literally an aggregate, you should think long and hard. The more future-proof path is always to give it a constructor, so that later you can reorder its fields or add new fields without an API break. (For types that don't form part of an API—internal implementation details—I don't claim it'll matter either way.) Observe the cautionary tale of `std::div_t`⁵.

12.4 Implicit is correct for containers and sequences

The previous item boils down to “Types that behave like C structs should get implicit constructors from their ‘fields.’” This item boils down to “Types that behave like C arrays should get implicit constructors from their ‘elements.’”

Every single-argument constructor from `std::initializer_list` should be non-explicit.

```
vector(std::initializer_list<value_type> il);
// non-explicit
```

This also, arguably, explains why `std::string` and `std::string_view` have implicit constructors from string literals. You could think of

```
std::string s = "hello world";
```

as “implicitly converting” a Platonic string datum from one C++ type to another; but arguably you could also think of it as “initializing” the characters of `s` with the characters of that string literal.

```
const char a[] = "hello world"; // same deal
```

Note that the ability to have “elements of a sequence” isn’t connected with ownership. A `string_` view can be constructed from a string, and a `span` or `initializer_list` can be constructed from a braced initializer.

C++20 `span` provides another cautionary tale: as of 2023, it is constructible from a braced initializer list, but only explicitly. So you can call `void f(span<const int>)` as `f(span<const int>({1,2,3}))` or `f(il)` or even `f({{1,2,3}})`, but not as `f({1,2,3})` directly (<https://godbolt.org/z/e4a5E5fGb>). P2447 is attempting to fix that retroactively. Meanwhile, try not to reproduce `span`'s mistake in your own code: initializer-list constructors should always be non-explicit.

Types constructible from `initializer_list` should also have implicit default constructors: a little-known quirk of C++ is that `A a = {};` will create a zero-element `initializer_list` if it must, but it'll prefer the default constructor if there is one. So if you intend that syntax to be well-formed, you should make sure your default constructor is either non-declared or non-explicit.

⁵<https://en.cppreference.com/w/cpp/numeric/math/div>

12.5 Implicit is correct for string and function

C++ types that deliberately set out to mimic other types should probably have non-**explicit** single-argument “converting constructors” from those other types. For example, it makes sense that `std::string` is implicitly convertible from `const char*`; that `std::function<int()>` is implicitly convertible from `int (*)()`; and that your own `BigInt` type might be implicitly convertible from `long long`.

Another way to think of this is: Types that represent the same Platonic domain should probably be implicitly interconvertible. For example, it makes sense that all integer types are implicitly interconvertible, and your `BigInt` type maybe should join them. But beware: usually when we have more than one C++ type, it’s because there’s a difference we’re trying to preserve. `string_view` is convertible from `string` because they’re both “strings” in the Platonic sense; but `string` is not implicitly convertible from `string_view` because when we’re writing code with `string_view` we don’t expect implicit memory allocations—it would be bad if our `string_view` quietly converted to `string` and allocated a bunch of memory when we weren’t looking. The same idea (but worse) explains why `int*` and `unique_ptr<int*>` don’t implicitly interconvert: We don’t want to quietly take ownership of, or quietly release or duplicate ownership of, a pointer.

In fact, `unique_ptr<int>` also has an invariant that `int*` doesn’t: an `int*` can point anywhere, but a `unique_ptr<int>` can only (reasonably) point to a heap allocation. Similarly, a `string` can hold any kind of contents, but a `std::regex` can only (reasonably) hold a regular expression—all regular expressions are strings, but not all strings are regular expressions. We wouldn’t want to write a string in the source code, and have some other part of the code quietly start treating it like a regular expression. So, `regex`’s constructor from `string`⁶ is correctly marked **explicit**.

```

1 void f(std::string_view); // #1
2 void f(std::regex);      // #2
3 void test() {
4     f("hello world"); // unambiguously #1
5 }
```

12.6 Explicit is correct for most everything else

The vast majority of constructors fall under “everything else.”

When you write a constructor, remember that omitting **explicit** will enable implicit conversions from whatever the constructor’s parameter types are. Then remind yourself that braced initialization should be used only for “elements.” In this constructor parameter list, do the parameters’ values correspond one-for-one with the “elements” of my object’s value?

```

struct MyRange {
    MyRange(int *first, int *last);
    MyRange(std::vector<int> initial_values);
};
```

⁶https://en.cppreference.com/w/cpp/regex/basic_regex/basic_regex

These two implicit constructors can't both be correct. The first enables implicit conversion from `{&i, &j};` the second enables implicit conversion from `{1, 2}`. A `MyRange` value might be a pair of pointer values, or it might be a sequence of integer values, but it can't possibly be both at once. Our default assumption should be that it is neither: we should declare both of these constructors `explicit` until we see a good reason why not.

12.6.1 `explicit operator bool() const`

You should never declare conversion operators (`operator T() const`) at all; but if you must break that resolution, it'll probably be for `operator bool`. You might think that conversion to `bool` is usually implicit, in contexts like `(s ? 1 : 2)`. But in fact C++ defines a special “contextual conversion” just for `bool`, making each of these cases happy to call your `explicit operator bool`:

```

1 struct S {
2     explicit operator bool() const;
3 };
4 S s;
5 if (s) // OK
6 int i = s ? 1 : 2; // OK
7 bool b1 = s; // Error
8 bool b2 = true && s; // OK
9 void f(bool); f(s); // Error

```

Therefore, `operator bool` should always be `explicit`; you'll lose no “bool-like” functionality (the “OK” lines), while preventing some unwanted implicit conversions (the “Error” lines).

12.7 A stab at a complete guideline

- `A(const A&)` and `A(A&&)` should always be implicit.
- `A(std::initializer_list<T>)` should always be implicit. If you have both `A(std::initializer_list<T>)` and `A()`, then `A()` should also be implicit. Example: `vector`.
- Whenever `std::tuple_size_v<A>` exists, the corresponding `A(X,Y,Z)` should be implicit. That is, the well-formedness of `auto [x,y,z] = a` should imply the well-formedness of `A a = {x,y,z}`. Examples: `pair`, `tuple`.
- Type-erasure types intended as drop-in replacements in APIs should have implicit constructor templates from the types they replace. Examples: `string_view`, `function`, `any`.
- Every other constructor (even the zero-argument constructor!) should be `explicit` or have a very well-understood domain-specific reason why not. Example: `string(const char*)`.
- `operator bool` should always be `explicit`. Other `operator` Ts probably need to be implicit in order to do their job; but you should prefer named getter methods, anyway.

In short: You're going to see the `explicit` keyword a lot, and there's nothing wrong with that. Treat it as a keyword that means “Look out! Here comes a constructor declaration!”

Part III

C++ 23

C++23 是目前最新的 C++ 标准，本书整理这部分时，标准尚未正式发布。

虽还没发布，但包含哪些特性去年就早已确定了，单独把 C++23 从 Modern C++ 这一 Part 中分离出来，就是专门介绍这些最新的特性。

由于是一些新特性，所以本 Part 的文章难度等级普遍较低。一是许多特性本来就简单，二是当前编译器尚不支持许多特性，无法进一步探索，三是特性刚出不久，探索时间不足。

本 Part 建议先读 *Overview of C++23 Features*，这是我为大多数特性写的一篇概览文章，速览一遍，知晓有哪些新加特性，再去阅读对应的特性文章，效果更好。

2023 年 5 月 13 日
里缪

Chapter 13

Overview of C++23 Features

• 里缪 2023-01-10



本文合并了我写过的两篇 C++23 主题的文章，覆盖绝大多数 C++23 的特性。

将这篇概述放在本主题的开篇，是为了帮大家构建一个完整的框架，若要更加深入地理解其中的某些特性，接着去看后面的其他文章即可。

13.1 Deducing this (P0847)

Deducing this 是 C++23 中最主要的特性之一。msvc 在去年 3 月份就已支持该特性，可以在 v19.32 之后的版本使用。

为什么我们需要这个特性？

大家知道，成员函数都有一个隐式对象参数，对于非静态成员函数，这个隐式对象参数就是 `this` 指针；而对于静态成员函数，这个隐式对象参数被定义为可以匹配任何参数，这仅仅是为了保证重载决议可以正常运行。

Deducing this 所做的事就是提供一种将非静态成员函数的「隐式对象参数」变为「显式对象参数」的方式。为何只针对非静态成员函数呢？因为静态成员函数并没有 `this` 指针，隐式对象参数并不能和 `this` 指针划等号，静态函数拥有隐式对象参数只是保证重载决议能够正常运行而已，这个参数没有其他用处。

于是，现在便有两种写法编写非静态成员函数：

```
1 struct S_implicit {
2     void foo() {}
3 };
4
5 struct S_explicit {
6     void foo(this S_explicit&) {}
7 };
```

通过 Deducing this，可以将隐式对象参数显式地写出来，语法为 `this+type`。该提案最根本的动机是消除成员函数修饰所带来的冗余，举个例子：

```
1 // Before
2 struct S_implicit {
```

```

3     int data_;
4
5     int& foo() & { return data_; }
6     const int& foo() const& { return data_; }
7 };
8
9 // After
10 struct S_explicit {
11     int data_;
12
13     template <class Self>
14     auto&& foo(this Self& self) {
15         return std::forward<Self>(self).data_;
16     }
17 };

```

原本你也许得为同一个成员函数编写各种版本的修饰，比如`&`, `const&`, `&&`, `const &&`，其逻辑并无太大变化，完全是重复的机械式操作。如今借助 Deducing this，你只需编写一个版本即可。

这里使用了模板形式的参数，通常来说，建议是使用`Self`作为显式对象参数的名称，顾名思义的同时又能和其他语言保持一致性。

该特性还有许多使用场景，同时也是一种新的定制点表示方式。比如，借助 Deducing this，可以实现递归 Lambdas。

```

1 int main() {
2     auto gcd = [] (this auto self, int a, int b) -> int {
3         return b == 0 ? a : self(b, a % b);
4     };
5
6     std::cout << gcd(20, 30) << "\n";
7 }

```

这使得 Lambda 函数再次得到增强。

又比如，借助 Deducing this，可以简化 CRTP。

```

1 //// Before
2 // CRTP
3 template <class Derived>
4 struct Base {
5     void foo() {
6         auto& self = *static_cast<Derived*>(this);
7         self.bar();
8     }
9 };
10
11 struct Derived : Base<Derived> {

```

```

12     void bar() const {
13         std::cout << "CRTP Derived\n";
14     }
15 };
16
17 ///////////////////////////////////////////////////////////////////
18 ////////////////////////////////////////////////////////////////// After
19 // Deducing this
20 struct Base {
21     template <class Self>
22     void foo(this Self& self) {
23         self.bar();
24     }
25 };
26
27 struct Derived : Base {
28     void bar() const {
29         std::cout << "Deducing this Derived\n";
30     }
31 };

```

这种新的方式实现 CRTP，可以省去 CR，甚至是 T，要更加自然，更加清晰。这也是一种新的定制点方式，稍微举个简单点的例子：

```

1 // Library
2 namespace mylib {
3
4     struct S {
5         auto abstract_interface(this auto& self, int param) {
6             self.concrete_algo1(self.concrete_algo2(param));
7         }
8     };
9 } // namespace mylib
10
11 namespace userspace {
12     struct M : mylib::S {
13         auto concrete_algo1(int val) {}
14         auto concrete_algo2(int val) const {
15             return val * 6;
16         }
17     };
18 } // namespace userspace
19
20 int main() {

```

```

21     using userspace::M;
22     M m;
23     m.abstract_interface(4);
24 }
```

这种方式依旧属于静态多态的方式，但代码更加清晰、无侵入，并支持显式 opt-in，是一种值得使用的方式。定制点并非一个简单的概念，若是看不懂以上例子，跳过便是。

下面再来看其他的使用场景。

Deducing this 还可以用来解决根据closure类型完美转发 Lambda 捕获参数的问题。

亦即，如果 Lambda 函数的类型为左值，那么捕获的参数就以左值转发；如果为右值，那么就以右值转发。下面是一个例子：

```

1 #include <iostream>
2 #include <type_traits>
3 #include <utility> // for std::forward_like
4
5 auto get_message() {
6     return 42;
7 }
8
9 struct Scheduler {
10     auto submit(auto&& m) {
11         std::cout << std::boolalpha;
12         std::cout << std::is_lvalue_reference<decltype(m)>::value << "\n";
13         std::cout << std::is_rvalue_reference<decltype(m)>::value << "\n";
14         return m;
15     }
16 };
17
18 int main() {
19     Scheduler scheduler;
20     auto callback = [m=get_message(), &scheduler](this auto&& self) -> bool {
21         return scheduler.submit(std::forward_like<decltype(self)>(m));
22     };
23     callback(); // retry(callback)
24     std::move(callback)(); // try-or-fail(rvalue)
25 }
26
27 // Output:
28 // true
29 // false
30 // false
31 // true
```

若是没有 Deducing this，那么将无法简单地完成这个操作。

另一个用处是可以将`this`以值形式传递，对于小对象来说，可以提高性能。一个例子：

```

1  struct S {
2      int data_;
3      int foo(); // implicit this pointer
4      // int foo(this S); // Pass this by value
5  };
6
7  int main() {
8      S s{42};
9      return s.foo();
10 }
11
12 // implicit this pointer 生成的汇编代码:
13 // sub    rsp, 40          ; 00000028H
14 // lea     rcx, QWORD PTR s$[rsp]
15 // mov    DWORD PTR s$[rsp], 42   ; 0000002aH
16 // call   int S::foo(void)       ; S::foo
17 // add    rsp, 40          ; 00000028H
18 // ret    0
19
20 // Pass this by value 生成的汇编代码:
21 // mov    ecx, 42          ; 0000002aH
22 // jmp    static int S::foo(this S) ; S::foo

```

对于隐式的`this`指针，生成的汇编代码需要先分配栈空间，保存`this`指针到`rcx`寄存器中，再将`42`赋值到`data_`中，然后调用`foo()`，最后平栈。

而以值形式传递`this`，则无需那些操作，因为值传递的`this`不会影响`s`变量，中间的步骤都可以被优化掉，也不再需要分配和平栈操作，所以可以直接将`42`保存到寄存器当中，再`jmp`到`foo()`处执行。

Deducing `this` 是个单独就可写篇四五星难度文章的特性，用处很多，值得深入探索的地方也很多，所以即便是概述这部分也写得比较多。Sy Brand 是该提案的作者之一，后面有一篇他的四星文章，大家可以阅读一下。

13.2 Monadic `std::optional` (P0798R8)

P0798 提议为`std::optional`增加三个新的成员：`map()`, `and_then()`和`or_else()`。

功能分别为：

- `map`: 对`optional`的值应用一个函数，返回`optional`中 wrapped 的结果。若是`optional`中没有值，返回一个空的`optional`；
- `and_then`: 组合使用返回`optional`的函数；
- `or_else`: 若是有值，返回`optional`；若是无值，则调用传入的函数，在此可以处理错误。

在 R2 中 `map()` 被重命名为 `transform()`, 因此实际新增的三个函数为 `transform()`, `and_then()` 和 `or_else()`。这些函数主要是避免手动检查 `optional` 值是否有效, 比如:

```

1 // Before
2 if (opt_string) {
3     std::optional<int> i = stoi(*opt_string);
4 }
5
6 // After
7 std::optional<int> i = opt_string.and_then(stoi);

```

一个使用的小例子:

```

1 // chain a series of functions until there's an error
2 std::optional<string> opt_string("10");
3 std::optional<int> i = opt_string
4             .and_then(std::stoi)
5             .transform([](auto i) { return i * 2; });

```

错误的情况:

```

1 // fails, transform not called, j == nullopt
2 std::optional<std::string> opt_string_bad("abcd");
3 std::optional<int> j = opt_string_bad
4             .and_then(std::stoi)
5             .transform([](auto i) { return i * 2; });

```

目前 GCC 12, Clang 14, MSVC v19.32 已经支持该特性。

13.3 std::expected (P0323)

该特性用于解决错误处理的问题, 增加了一个新的头文件 `<expected>`。

错误处理的逻辑关系为条件关系, 若正确, 则执行 A 逻辑; 若失败, 则执行 B 逻辑, 并需要知道确切的错误信息, 才能对症下药。当前的常用方式是通过错误码或异常, 但使用起来还是多有不便。

`std::expected<T, E>` 表示期望, 算是 `std::variant` 和 `std::optional` 的结合, 它要么保留 T (期望的类型), 要么保留 E (错误的类型), 它的接口又和 `std::optional` 相似。一个简单的例子:

```

1 enum class Status : uint8_t {
2     Ok,
3     connection_error,
4     no_authority,
5     format_error,
6 };
7
8 bool connected() {
9     return true;

```

```

10  }
11
12 bool has_authority() {
13     return false;
14 }
15
16 bool format() {
17     return false;
18 }
19
20 std::expected<std::string, Status> read_data() {
21     if (!connected())
22         return std::unexpected<Status> { Status::connection_error };
23     if (!has_authority())
24         return std::unexpected<Status> { Status::no_authority };
25     if (!format())
26         return std::unexpected<Status> { Status::format_error };
27
28     return {"my expected type"};
29 }
30
31
32 int main() {
33     auto result = read_data();
34     if (result) {
35         std::cout << result.value() << "\n";
36     } else {
37         std::cout << "error code: " << (int)result.error() << "\n";
38     }
39 }
```

这种方式无疑会简化错误处理的操作。

该特性目前在 GCC 12, Clang 16 (还未发布), MSVC v19.33 已经实现。

13.4 Multidimensional Arrays (P2128)

这个特性用于访问多维数组, 之前C++ `operator[]` 只支持访问单个下标, 无法访问多维数组。因此要访问多维数组, 以前的方式是:

- 重载`operator()`, 于是能够以`m(1, 2)` 来访问第 1 行第 2 个元素。但这种方式容易和函数调用产生混淆;
- 重载`operator[]`, 并以`std::initializer_list` 作为参数, 然后便能以`m[{1, 2}]` 来访问元素。但这种方式看着别扭;

- 链式链接**operator[]**，然后就能够以`m[1][2]`来访问元素。同样，看着别扭至极；
- 定义一个**at()**成员，然后通过`at(1, 2)`访问元素。同样不方便。

感谢该提案，在C++23，我们终于可以通过`m[1][2]`这种方式来访问多维数组。一个例子：

```

1 template <class T, size_t R, size_t C>
2 struct matrix {
3     T& operator[](const size_t r, const size_t c) noexcept {
4         return data_[r * C + c];
5     }
6
7     const T& operator[](const size_t r, const size_t c) const noexcept {
8         return data_[r * C + c];
9     }
10
11 private:
12     std::array<T, R * C> data_;
13 };
14
15
16 int main() {
17     matrix<int, 2, 2> m;
18     m[0, 0] = 0;
19     m[0, 1] = 1;
20     m[1, 0] = 2;
21     m[1, 1] = 3;
22
23     for (auto i = 0; i < 2; ++i) {
24         for (auto j = 0; j < 2; ++j) {
25             std::cout << m[i, j] << ' ';
26         }
27         std::cout << std::endl;
28     }
29 }
```

该特性目前在GCC 12和Clang 15以上版本已经支持。

13.5 if consteval (P1938)

该特性是关于**immediate function**的，即**consteval function**。

解决的问题其实很简单，在C++20，**consteval function**可以调用**constexpr function**，而反过来却不行。

```

1 consteval auto bar(int m) {
2     return m * 6;
```

```

3   }
4
5 constexpr auto foo(int m) {
6     return bar(m);
7 }
8
9 int main() {
10   [[maybe_unused]] auto res = foo(42);
11 }
```

以上代码无法编译通过，因为 **constexpr function** 不是强保证执行于编译期，在其中自然无法调用 **consteval function**。

但是，即便加上 **if std::is_constant_evaluated()** 也无法编译成功。

```

1 constexpr auto foo(int m) {
2   if (std::is_constant_evaluated()) {
3     return bar(m);
4   }
5   return 42;
6 }
```

这就存在问题了，P1938 通过 **if consteval** 修复了这个问题。在 C++23，可以这样写：

```

1 constexpr auto foo(int m) {
2   if consteval {
3     return bar(m);
4   }
5   return 42;
6 }
```

该特性目前在 GCC 12 和 Clang 14 以上版本已经实现。

13.6 Formatted Output (P2093)

该提案就是 **std::print()**，之前已经说过，这里再简单地说下。标准 cout 的设计非常糟糕，具体表现在：

- 可用性差，基本没有格式化能力；
- 会多次调用格式化 I/O 函数；
- 默认会同步标准 C，性能低；
- 内容由参数交替组成，在多线程环境，内容会错乱显示；
- 二进制占用空间大；
-

随着 **Formatting Library** 加入 C++20，已在 fmt 库中使用多年的 `fmt::print()` 加入标准也是顺理成章。

格式化输出的目标是要满足：可用性、Unicode 编码支持、良好的性能，与较小的二进制占用空间。为了不影响现有代码，该特性专门加了一个新的头文件 `<print>`，包含两个主要函数：

```

1 #include <print>
2
3 int main() {
4     const char* world = "world";
5     std::print("Hello {}", world);    // doesn't print a newline
6     std::println("Hello {}", world); // print a newline
7 }
```

这对 `cout` 来说绝对是暴击，`std::print` 的易用性和性能简直完爆它。其语法就是 **Formatting Library** 的格式化语法，可参考 Using C++20 Formatting Library¹。性能对比：

Benchmark	Time	CPU	Iterations
<hr/>			
printf	87.0 ns	86.9 ns	7834009
ostream	255 ns	255 ns	2746434
print	78.4 ns	78.3 ns	9095989
print_cout	89.4 ns	89.4 ns	7702973
print_cout_sync	91.5 ns	91.4 ns	7903889

结果显示，`printf` 与 `print` 几乎要比 `cout` 快三倍，`print` 默认会打印到 `stdout`。当打印到 `cout` 并同步标准 C 的流时（`print_cout_sync`），`print` 大概要快 14%；当不同步标准 C 的流时（`print_cout`），依旧要快不少。

遗憾的是，该特性目前没有编译器支持。

13.7 Formatting Ranges (P2286)

同样属于 **Formatting** 大家族，该提案使得我们能够格式化输出 **Ranges**。也就是说，我们能够写出这样的代码：

```

1 import std;
2
3 auto main() -> int {
4     std::vector vec { 1, 2, 3 };
5     std::print("{}\n", vec); // Output: [1, 2, 3]
6 }
```

这意味着再也不用迭代来输出 **Ranges** 了。

这是非常有必要的，考虑一个简单的需求：文本分割。

Python 的实现：

¹shorturl.at/nuvxM

```

1 print("how you doing".split(" "))
2
3 # Output:
4 # ['how', 'you', 'doing']

```

Java 的实现:

```

1 import java.util.Arrays;
2
3 class Main {
4     public static void main(String args[]) {
5         System.out.println("how you doing".split(" "));
6         System.out.println(Arrays.toString("how you doing".split(" ")));
7     }
8 }
9
10 // Output:
11 // [Ljava.lang.String;@2b2fa4f7
12 // [how, you, doing]

```

Rust 的实现:

```

1 use itertools::Itertools;
2
3 fn main() {
4     println!("{:?}", "How you doing".split(' '));
5     println!("[{}]", "How you doing".split(' ').format(", "));
6     println!("{:?}", "How you doing".split(' ').collect::<Vec<_>>());
7 }
8
9 // Output:
10 // Split(SplitInternal { start: 0, end: 13, matcher:
11 //   CharSearcher { haystack: "How you doing", finger: 0, finger_back: 13, needle: ' ',
12 //     utf8_size: 1, utf8_encoded: [32, 0, 0, 0] }, allow_trailing_empty: true, finished: false })
13 // [How, you, doing]
14 // ["How", "you", "doing"]

```

JS 的实现:

```

1 console.log('How you doing'.split(' '))
2
3 // Output:
4 // ["How", "you", "doing"]

```

Go 的实现:

```

1 package main
2 import "fmt"

```

```

3 import "strings"
4
5 func main() {
6     fmt.Println(strings.Split("How you doing", " "));
7 }
8
9 // Output:
10 // [How you doing]

```

Kotlin 的实现:

```

1 fun main() {
2     println("How you doing".split(" "));
3 }
4
5 // Output:
6 // [How, you, doing]

```

C++ 的实现:

```

1 int main() {
2     std::string_view contents {"How you doing"};
3
4     auto words = contents
5         | std::views::split(' ')
6         | std::views::transform([](auto&& str) {
7             return std::string_view(&str.begin(), std::ranges::distance(str));
8         });
9
10    std::cout << "[";
11    char const* delim = "";
12    for (auto word : words) {
13        std::cout << delim;
14
15        std::cout << std::quoted(word);
16        delim = ", ";
17    }
18    std::cout << "]\\n";
19 }
20
21 // Output:
22 // ["How", "you", "doing"]

```

借助 fmt, 可以简化代码:

```

1 int main() {
2     std::string_view contents {"How you doing"};

```

```

3
4     auto words = contents
5         | std::views::split(' ')
6         | std::views::transform([](auto&& str) {
7             return std::string_view(&str.begin(), std::ranges::distance(str));
8         });
9
10    fmt::print("{}\n", words);
11    fmt::print("<<{}>>", fmt::join(words, "--"));
12
13 }
14
15 // Output:
16 // ["How", "you", "doing"]
17 // <<How--you--doing>>

```

因为`views::split()`返回的是一个 subrange，因此需要将其转变成`string_view`，否则，输出将为：

```

1 int main() {
2     std::string_view contents {"How you doing"};
3
4     auto words = contents | std::views::split(' ');
5
6     fmt::print("{}\n", words);
7     fmt::print("<<{}>>", fmt::join(words, "--"));
8
9 }
10
11 // Output:
12 // [[H, o, w], [y, o, u], [d, o, i, n, g]]
13 // <<['H', 'o', 'w']--['y', 'o', 'u']--['d', 'o', 'i', 'n', 'g']>>

```

总之，这个特性将极大简化 Ranges 的输出，是值得兴奋的特性之一。

该特性目前没有编译器支持。

13.8 import std (P2465)

C++20 模块很难用的一个原因就是标准模块没有提供，因此这个特性的加入是自然趋势。

现在，可以写出这样的代码：

```

1 import std;
2
3 int main() {
4     std::print("Hello standard library modules!\n");
5 }

```

性能对比:

	#include needed headers	Import needed headers	import std	#include all headers	Import all headers
”Hello world” (<iostream>)	0.87s	0.32s	0.08s	3.43s	0.62s
”Mix” (9 headers)	2.20s	0.77s	0.44s	3.53s	0.99s

如果你是混合 C 和 C++, 那可以使用 `std.compat module`, 所有的 C 函数和标准库函数都会包含进来。

目前基本没有编译器支持此特性。

13.9 out_ptr (P1132r8)

23 新增了两个对于指针的抽象类型, `std::out_ptr_t` 和 `std::inout_ptr_t`, 两个新的函数 `std::out_ptr()` 和 `std::inout_ptr()` 分别返回这两个类型。

主要是在和 C API 交互时使用的, 一个例子对比一下:

```

1 // Before
2 int old_c_api(int**);
3
4 int main() {
5     auto up = std::make_unique<int>(5);
6
7     int* up_raw = up.release();
8     if (int ec = foreign_resetter(&up)) {
9         return ec;
10    }
11
12    up.reset(up_raw);
13 }
14
15 /////////////////////////////////
16 // After
17 int old_c_api(int**);
18
19 int main() {
20     auto up = std::make_unique<int>(5);
21
22     if (int ec = foreign_resetter(std::inout_ptr(up))) {
23         return ec;
24    }
25
26    // *up is still valid

```

27

该特性目前在 MSVC v19.30 支持。

13.10 `auto(x) decay copy` (P0849)

该提案为`auto`又增加了两个新语法：`auto(x)` 和 `auto{x}`。两个作用一样，只是写法不同，都是为 `x` 创建一份拷贝。

为什么需要这么个东西？看一个例子：

```

1 void bar(const auto&);

2

3 void foo(const auto& param) {
4     auto copy = param;
5     bar(copy);
6 }
```

`foo()` 中调用 `bar()`，希望传递一份 `param` 的拷贝，则我们需要单独多声明一个临时变量。或是这样：

```

void foo(const auto& param) {
    bar(std::decay_t<decltype(param)>{param});
}
```

这种方式需要手动去除多余的修饰，只留下 `T`，要更加麻烦。

`auto(x)` 就是内建的 **decay copy**，现在可以直接这样写：

```

void foo(const auto& param) {
    bar(auto{param});
}
```

大家可能还没意识到其必要性，来看提案当中更加复杂一点的例子。

```

1 void pop_front_alike(auto& container) {
2     std::erase(container, container.front());
3 }

4

5 int main() {
6     std::vector fruits{ "apple", "apple", "cherry", "grape",
7         "apple", "papaya", "plum", "papaya", "cherry", "apple"};
8     pop_front_alike(fruits);
9

10    fmt::print("{}\n", fruits);
11 }

12

13 // Output:
14 // ["cherry", "grape", "apple", "papaya", "plum", "papaya", "apple"]
```

请注意该程序的输出,是否如你所想的一样。若没有发现问题,请容许我再提醒一下:`pop_front_alike()`要移除容器中所有跟第 1 个元素相同的元素。

因此,理想的结果应该为:

```
["cherry", "grape", "papaya", "plum", "papaya", "cherry"]
```

是哪里出了问题呢?让我们来看看 gcc `std::erase()`的实现:

```

1  template<typename _ForwardIterator, typename _Predicate>
2  _ForwardIterator
3      __remove_if(_ForwardIterator __first, _ForwardIterator __last,
4          _Predicate __pred)
5  {
6      __first = std::__find_if(__first, __last, __pred);
7      if (__first == __last)
8          return __first;
9      _ForwardIterator __result = __first;
10     ++__first;
11     for (; __first != __last; ++__first)
12         if (!__pred(__first)) {
13             *____result = __GLIBCXX_MOVE(*__first);
14             ++__result;
15         }
16
17     return __result;
18 }
19
20 template<typename _Tp, typename _Alloc, typename _Up>
21     inline typename vector<_Tp, _Alloc>::size_type
22     erase(vector<_Tp, _Alloc>& __cont, const _Up& __value)
23 {
24     const auto __osz = __cont.size();
25     __cont.erase(std::remove(__cont.begin(), __cont.end(), __value),
26                 __cont.end());
27     return __osz - __cont.size();
28 }
```

`std::remove()`最终调用的是`remove_if()`,因此关键就在这个算法里面。这个算法每次会比较当前元素和欲移除元素,若不相等,则用当前元素覆盖当前`_result`迭代器的值,然后`_result`向后移一位。重复这个操作,最后全部有效元素就都跑到`_result`迭代器的前面去了。

问题出在哪里呢?欲移除元素始终指向首个元素,而它会随着元素覆盖操作被改变,因为它的类型为`const T&`。

此时,必须重新 copy 一份值,才能得到正确的结果。

故将代码小作更改,就能得到正确的结果。

```
void pop_front_alike(auto& container) {
    auto copy = container.front();
    std::erase(container, copy);
}
```

然而这种方式是非常反直觉的，一般来说这两种写法的效果应该是等价的。我们将 `copy` 定义为一个单独的函数，表达效果则要好一点。

```
1 auto copy(const auto& value) {
2     return value;
3 }
4
5 void pop_front_alike(auto& container) {
6     std::erase(container, copy(container.front()));
7 }
```

而 `auto{x}` 和 `auto(x)`，就相当于这个 `copy()` 函数，只不过它是内建到语言里面的而已。

13.11 Narrowing contextual conversions to bool (P1401R5)

这个提案允许在 `static_assert` 和 `if constexpr` 中从整形转换为布尔类型。

以下表格就可以表示所有内容。

Before	After
<code>if constexpr(bool(flags & Flags::Exec))</code>	<code>if constexpr(flags & Flags::Exec)</code>
<code>if constexpr(flags & Flags::Exec != 0)</code>	<code>if constexpr(flags & Flags::Exec)</code>
<code>static_assert(N % 4 != 0);</code>	<code>static_assert(N % 4);</code>
<code>static_assert(bool(N));</code>	<code>static_assert(N);</code>

对于严格的 C++ 编译器来说，以前在这种情境下 `int` 无法向下转换为 `bool`，需要手动强制转换，C++23 这一情况得到了改善。

目前在 GCC 9 和 Clang 13 以上版本支持该特性。

13.12 forward_like (P2445)

这个在 **Deducing this** 那节已经使用过了，是同一个作者。使用情境让我们回顾一下这个例子：

```
1 auto callback = [m = get_message(), &scheduler](this auto&& self) -> bool {
2     return scheduler.submit(std::forward_like<decltype(self)>(m));
3 };
4
5 callback();           // retry(callback)
6 std::move(callback()); // try-or-fail(rvalue)
```

`std::forward_like` 加入到了 `<utility>` 中，就是根据模板参数的值类别来转发参数。如果 **closure type** 为左值，那么 `m` 将转发为左值；如果为右值，将转发为右值。

听说 Clang 16 和 MSVC v19.34 支持该特性，但都尚未发布。

13.13 #elifdef and #elifndef (P2334)

这两个预处理指令来自 WG14 (C 的工作组), 加入到了 C23。C++ 为了兼容 C, 也将它们加入了 C++23。

也是一个完善工作。`#ifdef` 和 `#ifndef` 分别是 `#if defined()` 和 `#if !defined()` 的简写, 而 `#_elif defined()` 和 `#_elif !defined()` 却并没有与之对应的简写指令。因此, C23 使用 `#elifdef` 和 `#elifndef` 来补充这一遗漏。

总之, 是两个非常简单的小特性。目前已在 GCC 12 和 Clang 13 得到支持。

13.14 #warning (P2437)

`#warning` 是主流编译器都会支持的一个特性, 最终倒逼 C23 和 C++23 也加入了进来。

这个小特性可以用来产生警告信息, 与 `#error` 不同, 它并不会停止翻译。

用法很简单:

```
#ifndef FOO
#warning "FOO defined, performance might be limited"
#endif
```

目前除了 MSVC 不支持该特性, 其他主流编译器都支持。

13.15 constexpr std::unique_ptr (P2273R3)

`std::unique_ptr` 也支持编译期计算了, 一个小例子:

```
1 constexpr auto fun() {
2     auto p = std::make_unique<int>(4);
3     return *p;
4 }
5
6 int main() {
7     constexpr auto i = fun();
8     static_assert(4 == i);
9 }
```

目前 GCC 12 和 MSVC v19.33 支持该特性。

13.16 Improving string and string_view (P1679R3, P2166R1, P1989R2, P1072R10, P2251R1)

`string` 和 `string_view` 也获得了一些增强, 这里简单地说下。

P1679 为二者增加了一个 `contains()` 函数, 小例子:

```
std::string str("dummy text");
if (str.contains("dummy")) {
```

```
// do something
}
```

目前 GCC 11, Clang 12, MSVC v19.30 支持该特性。

P2166 使得它们从`nullptr`构建不再产生 UB, 而是直接编译失败。

```
std::string s { nullptr }; // error!
std::string_view sv { nullptr }; // error!
```

目前 GCC 12, Clang 13, MSVC v19.30 支持该特性。

P1989 是针对`std::string_view`的, 一个小例子搞定:

```
1 int main() {
2     std::vector v { 'a', 'b', 'c' };
3
4     // Before
5     std::string_view sv(v.begin(), v.end());
6
7     // After
8     std::string_view sv23 { v };
9 }
```

以前无法直接从 Ranges 构建`std::string_view`, 而现在支持这种方式。

该特性在 GCC 11, Clang 14, MSVC v19.30 已经支持。

P1072 为`string`新增了一个成员函数:

```
template< class Operation >
constexpr void resize_and_overwrite( size_type count, Operation op );
```

可以通过提案中的一个示例来理解:

```
1 int main() {
2     std::string s { "Food: " };
3
4     s.resize_and_overwrite(10, [](char* buf, int n) {
5         return std::find(buf, buf + n, ':') - buf;
6     });
7
8     std::cout << std::quoted(s) << '\n'; // "Food"
9 }
```

主要是两个操作: 改变大小和覆盖内容。第1个参数是新的大小, 第2个参数是一个`op`, 用于设置新的内容。

然后的逻辑是 (`maxsize`就是第一个参数):

- 如果`maxsize <= s.size()`, 删除最后的`s.size()-maxsize`个元素;
- 如果`maxsize > s.size()`, 追加`maxsize-size()`个默认元素;

- 调用`erase(begin() + op(data(), maxsize), end())`。

这里再给出一个例子，可以使用上面的逻辑来走一遍，以更清晰地理解该函数。

```

1 constexpr std::string_view fruits[] {"apple", "banana", "coconut", "date", "elderberry"};
2 std::string s1 { "Food: " };
3
4 s1.resize_and_overwrite(16, [sz = s1.size()](char* buf, std::size_t buf_size) {
5     const auto to_copy = std::min(buf_size - sz, fruits[1].size()); // 6
6     std::memcpy(buf + sz, fruits[1].data(), to_copy); // append "banana" to s1.
7     return sz + to_copy; // 6 + 6
8 });
9
10 std::cout << s1; // Food: banana

```

注意一下，`maxsize`是最大的可能大小，而`op`返回才是实际大小，因此逻辑的最后才有一个`erase()`操作，用于删除多余的大小。

这个特性在 GCC 12, Clang 14, MSVC v19.31 已经实现。

接着来看 P2251，它更新了`std::span`和`std::string_view`的约束，从 C++23 开始，它们必须满足 **TriviallyCopyable Concept**。主流编译器都支持该特性。

最后来看 P0448，其引入了一个新的头文件`<spanstream>`。

大家都知道，`stringstream`现在被广泛使用，可以将数据存储到`string`或`vector`当中，但这些容器当数据增长时会发生「挪窝」的行为，若是不想产生这个开销呢？`<spanstream>`提供了一种选择，你可以指定固定大小的 buffer，它不会重新分配内存，但要小心数据超出 buffer 大小，此时内存的所有权在程序员这边。

一个小例子：

```

1 #define ASSERT_EQUAL(a, b) assert(a == b)
2 #define ASSERT(a) assert(a)
3
4 int main() {
5     char input[] = "10 20 30";
6     std::ispanstream is{ std::span<char>{input} };
7     int i;
8
9     is >> i;
10    ASSERT_EQUAL(10, i);
11
12    is >> i;
13    ASSERT_EQUAL(20, i);
14
15    is >> i;
16    ASSERT_EQUAL(30, i);
17
18    is >> i;

```

```

19     ASSERT(!is);
20 }
```

目前 GCC 12 和 MSVC v19.31 已支持该特性。

13.17 static operator() (P1169R4)

因为函数对象，Lambdas 使用得越来越多，经常作为标准库的定制点使用。这种函数对象只有一个 `operator()`，如果允许声明为 `static`，则可以提高性能。

至于原理，大家可以回顾一下 **Deducing this** 那节的 Pass this by value 提高性能的原理。明白静态函数和非静态函数在重载决议中的区别，大概就能明白这点。

顺便一提，由于 `multidimensional operator[]` 如今已经可以达到和 `operator()` 一样的效果，它也可以作为一种新的函数语法，你完全可以这样调用 `foo[]`，只是不太直观。因此，P2589 也提议了 `static operator[]`。

13.18 std::unreachable (P0627R6)

当我们知道某个位置是不可能执行到，而编译器不知道时，使用 `std::unreachable` 可以告诉编译器，从而避免没必要的运行期检查。

一个简单的例子：

```

1 void foo(int a) {
2     switch (a) {
3         case 1:
4             // do something
5             break;
6         case 2:
7             // do something
8             break;
9         default:
10            std::unreachable();
11    }
12 }
13
14 bool is_valid(int a) {
15     return a == 1 || a == 2;
16 }
17
18 int main() {
19     int a = 0;
20     while (!is_valid(a))
21         std::cin >> a;
22     foo(a);
23 }
```

该特性位于`<utility>`，在 GCC 12, Clang 15 和 MSVC v19.32 已经支持。

13.19 `std::to_underlying` (P1682R3)

同样位于`<utility>`，用于枚举到其潜在的类型，相当于以下代码的语法糖：

```
static_cast<std::underlying_type_t<Enum>>(e);
```

一个简单的例子就能看懂：

```
1 void print_day(int a) {
2     fmt::print("{}\n", a);
3 }
4
5 enum class Day : std::uint8_t {
6     Monday = 1,
7     Tuesday,
8     Wednesday,
9     Thursday,
10    Friday,
11    Saturday,
12    Sunday
13 };
14
15
16 int main() {
17     // Before
18     print_day(static_cast<std::underlying_type_t<Day>>(Day::Monday));
19
20     // C++23
21     print_day(std::to_underlying(Day::Friday));
22 }
```

的确很简单吧！该特性目前在 GCC 11, Clang 13, MSVC v19.30 已经实现。

13.20 `std::byteswap` (P1272R4)

位于`<bit>`，顾名思义，是关于位操作的。同样，一个例子看懂：

```
1 template <std::integral T>
2 void print_hex(T v)
3 {
4     for (std::size_t i = 0; i < sizeof(T); ++i, v >>= 8)
5     {
6         fmt::print("{} ", static_cast<unsigned>(T(0xFF) & v));
```

```

7     }
8     std::cout << '\n';
9 }
10
11 int main()
12 {
13     unsigned char a = 0xBA;
14     print_hex(a);           // BA
15     print_hex(std::byteswap(a)); // BA
16     unsigned short b = 0xBAAD;
17     print_hex(b);           // AD BA
18     print_hex(std::byteswap(b)); // BA AD
19     int c = 0xBAADFOOD;
20     print_hex(c);           // OD FO AD BA
21     print_hex(std::byteswap(c)); // BA AD FO OD
22     long long d = 0xBAADFOODBAADC0FE;
23     print_hex(d);           // FE CO AD BA OD FO AD BA
24     print_hex(std::byteswap(d)); // BA AD FO OD BA AD CO FE
25 }
```

可以看到，其作用是逆转整型的字节序。当需要在两个不同的系统传输数据，它们使用不同的字节序时（大端小端），这个工具就会很有用。

该特性目前在 GCC 12，Clang 14 和 MSVC v19.31 已经支持。

13.21 std::stacktrace (P0881R7, P2301R1)

位于`<stacktrace>`，可以让我们捕获调用栈的信息，从而知道哪个函数调用了当前函数，哪个调用引发了异常，以更好地定位错误。一个小例子：

```

1 void foo() {
2     auto trace = std::stacktrace::current();
3     std::cout << std::to_string(trace) << '\n';
4 }
5
6 int main() {
7     foo();
8 }
```

输出如下：

```

0# foo() at /app/example.cpp:5
1#      at /app/example.cpp:10
2#      at :0
3#      at :0
4#
```

注意，目前 GCC 12.1 和 MSVC v19.34 支持该特性，GCC 编译时要加上`-fstdc++_libbacktrace`参数。

`std::stacktrace`是`std::basic_stacktrace`使用默认分配器时的别名，定义为：

```
using stacktrace = std::basic_stacktrace<std::allocator<std::stacktrace_entry>>;
```

而 P2301，则是为其添加了 PMR 版本的别名，定义为：

```
namespace pmr {
    using stacktrace =
        std::basic_stacktrace<std::pmr::polymorphic_allocator<std::stacktrace_entry>>;
}
```

于是使用起来就会方便一些。

```

1 // Before
2 char buffer[1024];
3
4 std::pmr::monotonic_buffer_resource pool{
5     std::data(buffer), std::size(buffer)};
6
7 std::basic_stacktrace<
8     std::pmr::polymorphic_allocator<std::stacktrace_entry>>
9     trace{&pool};
10
11 // After
12 char buffer[1024];
13
14 std::pmr::monotonic_buffer_resource pool{
15     std::data(buffer), std::size(buffer)};
16
17 std::pmr::stacktrace trace{&pool};
```

这个特性到时再单独写篇文章，在此不细论。

13.22 Attributes (P1774R8, P2173R1, P2156R1)

Attributes 在 C++23 也有一些改变。

首先，P1774 新增了一个 Attribute `[[assume]]`，其实在很多编译器早已存在相应的特性，例如`__assume()`(MSVC, ICC)，`__builtin_assume()`(Clang)。GCC 没有相关特性，所以它也是最早实现标准`[[assume]]`的，目前就 GCC 13 支持该特性（等四月发布，该版本对 Rangs 的支持也很完善）。

现在可以通过宏来玩：

```

1 #if defined(__clang__)
2     #define ASSUME(expr) __builtin_assume(expr)
3 #elif defined(__GNUC__) && !defined(__ICC)
```

```

4     #define ASSUME(expr) if (expr) {} else { __builtin_unreachable(); }
5 #elif defined(_MSC_VER) || defined(__ICC)
6     #define ASSUME(expr) __assume(expr)
7 #endif

```

论文当中的一个例子：

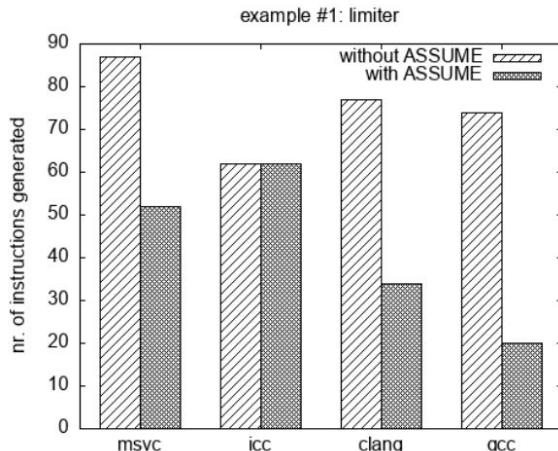
```

1 void limiter(float* data, size_t size) {
2     ASSUME(size > 0);
3     ASSUME(size % 32 == 0);
4
5     for (size_t i = 0; i < size; ++i) {
6         ASSUME(std::isfinite(data[i]));
7         data[i] = std::clamp(data[i], -1.0f, 1.0f);
8     }
9 }

```

第一个是假设 `size` 永不为 0，总是正数；第二个告诉编译器 `size` 总是 32 的倍数；第三个表明数据不是 NaN 或无限小数。这些假设不会被评估，也不会被检查，编译器假设其为真，依此优化代码。若是假设为假，可能会产生 UB。

使用该特性与否编译产生的指令数对比结果如下图。



其次，P2173 使得可以在 Lambda 表达式上使用 Attributes，一个例子：

```

1 // Any attribute so specified does not appertain to the function
2 // call operator or operator template itself, but its type.
3 auto lam = [] [[nodiscard]] -> int { return 42; };
4
5 int main()
6 {
7     lam();
8 }
9
10 // Output:

```

```

11 // <source>: In function 'int main()':
12 // <source>:12:8: warning: ignoring return value of '<lambda()>',
13 //                         declared with attribute 'nodiscard' [-Wunused-result]
14 //     12 |     lam();
15 //     | ~~~^~_
16 // <source>:8:12: note: declared here
17 //     8 | auto lam = [] [[nodiscard]] ->int { return 42; };
18 //     |

```

注意，Attributes 属于 **closure type**，而不属于 **operator ()**。因此，有些 Attributes 不能使用，比如 **[[noreturn]]**，它表明函数的控制流不会返回到调用方，而对于 Lambda 函数是会返回的。

除此之外，此处我还展示了 C++ 的另一个 Lambda 特性。在 C++23 之前，最简单的 Lambda 表达式为 **[]()**，而到了 C++23，则是 **[]**，可以省略无参时的括号，这得感谢 P1102。早在 GCC 9 就支持 **Attributes Lambda**，Clang 13 如今也支持。

最后来看 P2156，它移除了重复 Attributes 的限制。简单来说，两种重复 Attributes 的语法规判不一致。例子：

```

1 // Not allow
2 [[nodiscard, nodiscard]] auto foo() {
3     return 42;
4 }
5
6 // Allowed
7 [[nodiscard]][[nodiscard]] auto foo() {
8     return 42;
9 }

```

为了保证一致性，去除此限制，使得标准更简单。

什么时候会出现重复 Attributes，看论文怎么说：

During this discussion, it was brought up that the duplication across attribute-specifiers are to support cases where macros are used to conditionally add attributes to an attribute-specifier-seq, however it is rare for macros to be used to generate attributes within the same attribute-list. Thus, removing the limitation for that reason is unnecessary.

在基于宏生成的时候可能会出现重复 Attributes，因此允许第二种方式；宏生成很少使用第一种形式，因此标准限制了这种情况。但这却并没有让标准变得更简单。因此，最终移除了该限制。

目前使用 GCC 11，Clang 13 以上两种形式的结果将保持一致。

13.23 Lambdas (P1102R2, P2036R3, P2173R1)

Lambdas 表达式在 C++23 也再次迎来了一些新特性。像是支持 Lambda Attributes，可以省略 **()**，这在 Attributes 这一节已经介绍过，不再赘述。

另一个新特性是 P2036 提的，接下来主要说说这个。这个特性改变了 **trailing return types** 的 **Name Lookup** 规则，为什么？让我们来看一个例子。

```

1 double j = 42.0;
2 // ...
3 auto counter = [j = 0]() mutable -> decltype(j) {
4     return j++;
5 };

```

`counter`最终的类型是什么？是`int`吗？还是`double`？其实是`double`。

无论捕获列表当中存在什么值，**trailing return type** 的 Name Lookup 都不会查找到它。这意味着单独这样写将会编译出错：

```

1 auto counter = [j=0]() mutable -> decltype(j) {
2     return j++;
3 };
4
5 // Output:
6 // <source>:6:44: error: use of undeclared identifier 'j'
7 // auto counter = [j=0]() mutable -> decltype(j) {
8     ^
// 
```

因为对于 **trailing return type** 来说，根本就看不见捕获列表中的`j`。以下例子能够更清晰地展示这个错误：

```

1 template <typename T> int bar(int&, T&&);           // #1
2 template <typename T> void bar(int const&, T&&); // #2
3
4 int i;
5 auto f = [=](auto&& x) -> decltype(bar(i, x)) {
6     return bar(i, x);
7 }
8
9 f(42); // error

```

在 C++23，**trailing return types** 的 Name Lookup 规则变为：在外部查找之前，先查找捕获列表，从而解决这个问题。

目前没有任何编译器支持该特性。

13.24 Literal suffixes for (signed) `size_t` (P0330R8)

这个特性为`std::size_t`增加了后缀`uz`，为`signed std::size_t`加了后缀`z`。有什么用呢？看一个例子：

```

1 #include <vector>
2
3 int main() {
4     std::vector<int> v{0, 1, 2, 3};
5     for (auto i = 0uz, s = v.size(); i < s; ++i) {
```

```

6     /* use both i and v[i] */
7 }
8 }
```

这代码在 32 bit 平台编译能够通过，而放到 64 bit 平台编译，则会出现错误：

```

<source>(5): error C3538: in a declarator-list 'auto' must always deduce to the same type
<source>(5): note: could be 'unsigned int'
<source>(5): note: or      'unsigned __int64'
```

在 32 bit 平台上，`i`被推导为 `unsigned int`，`v.size()`返回的类型为 `size_t`。而 `size_t` 在 32 bit 上为 `unsigned int`，在 64 bit 上为 `unsigned long long`。(in MSVC) 因此，同样的代码，从 32 bit 切换到 64 bit 时就会出现错误。

通过新增的后缀，则可以保证这个代码在任何平台上都能有相同的结果。

```

1 #include <vector>
2
3 int main() {
4     std::vector<int> v{0, 1, 2, 3};
5     for (auto i = 0uz, s = v.size(); i < s; ++i) {
6         /* use both i and v[i] */
7     }
8 }
```

如此一来就解决了这个问题。目前 GCC 11 和 Clang 13 支持该特性。

13.25 std::mdspan (P0009r18)

`std::mdspan` 是 `std::span` 的多维版本，因此它是一个多维 Views。看一个例子，简单了解其用法。

```

1 int main()
2 {
3     std::vector v = {1,2,3,4,5,6,7,8,9,10,11,12};
4
5     // View data as contiguous memory representing 2 rows of 6 ints each
6     auto ms2 = std::experimental::mdspan(v.data(), 2, 6);
7     // View the same data as a 3D array 2 x 3 x 2
8     auto ms3 = std::experimental::mdspan(v.data(), 2, 3, 2);
9
10    // write data using 2D view
11    for(size_t i=0; i != ms2.extent(0); i++)
12        for(size_t j=0; j != ms2.extent(1); j++)
13            ms2[i, j] = i*1000 + j;
14
15    // read back using 3D view
16    for(size_t i=0; i != ms3.extent(0); i++)
```

```

17   {
18     fmt::print("slice @ i = {}\\n", i);
19     for(size_t j=0; j != ms3.extent(1); j++)
20     {
21       for(size_t k=0; k != ms3.extent(2); k++)
22         fmt::print("{} ", ms3[i, j, k]);
23       fmt::print("\\n");
24     }
25   }
26 }
```

目前没有编译器支持该特性，使用的是 <https://raw.githubusercontent.com/kokkos/mdspan/single-header/mdspan.hpp> 实现的版本，所以在 experimental 下面。`ms2` 是将数据以二维形式访问，`ms3` 则以三维访问，Views 可以改变原有数据，因此最终遍历的结果为：

```

1 slice @ i = 0
2 0 1
3 2 3
4 4 5
5 slice @ i = 1
6 1000 1001
7 1002 1003
8 1004 1005
```

这个特性值得剖析下其设计，这里不再深究，后面单独出一篇文章。

13.26 flat_map, flat_set (P0429R9, P1222R4)

C++23 多了 flat version 的 map 和 set：

- flat_map
- flat_set
- flat_multimap
- flat_multiset

过去的容器，有的使用二叉树，有的使用哈希表，而 flat 版本的使用的连续序列的容器，更像是容器的适配器。

无非就是时间或空间复杂度的均衡，目前没有具体测试，也没有编译器支持，暂不深究。

13.27 ranges::to (P1206R7)

C++20 中，Ranges 可以通过容器直接构造，而反过来却不行。

```

1 auto view = std::views::iota(0, 10) | std::views::common;
2 // std::vector<int> vec { view }; // ERROR!
3 std::vector<int> vec { std::ranges::begin(view), std::ranges::end(view) }; // OK
4
5 fmt::print("vec: {}\n", vec); // vec: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

甚至容器与容器之间也无法直接转换，只能这样使用：

```

1 std::list l { 1, 2, 3 };
2 // std::vector<decltype(l)::value_type> v { l }; // ERROR!
3 std::vector<decltype(l)::value_type> v { std::begin(l), std::end(l) }; // OK
4
5 fmt::print("v: {}\n", v); // v: [1, 2, 3]

```

而到了 C++23，引入了 `ranges::to`，可以方便地进行上述转换：

```

1 // views to container
2 auto view = views::iota(0, 10);
3 std::vector<int> vec = view | ranges::to<std::vector>();
4 fmt::print("view to vector: {}\n", vec);
5
6 // container to container
7 std::list l { 1, 2, 3 };
8 std::vector<decltype(l)::value_type> v = l | ranges::to<std::vector>();
9 fmt::print("list to vector: {}\n", v);
10
11 // Output:
12 // view to vector: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
13 // list to vector: [1, 2, 3]

```

13.28 Ranges: starts_with and ends_with (P1659)

C++20 为 `string` 和 `string_view` 引入了 `starts_with` 和 `ends_with` 算法，使得字符串匹配操作更加简便。一个小例子：

```

1 constexpr std::string_view str = "START_FLAG12345END_FLAG";
2 constexpr bool start_with_flag = str.starts_with("START_FLAG");
3 constexpr bool end_with_flag   = str.ends_with("END_FLAG");
4
5 fmt::print("start_with_flag: {}\nend_with_flag:{}\n", start_with_flag, end_with_flag);
6
7 // Output:
8 // start_with_flag: true
9 // end_with_flag:true

```

到了 C++23，Ranges 也加入这两个算法，使得能够对所有的容器进行该操作。同样一个小例子：

```

1 auto some_ints      = views::iota(0, 50);
2 auto some_more_ints = views::iota(0, 30);
3 if (ranges::starts_with(some_ints, some_more_ints)) {
4     fmt::print("starts_with true\n");
5 }
6
7 // Output:
8 // starts_with true

```

13.29 Views: zip, zip_transform, adjacent, and adjacent_transform (P2321R2)

这四个 Views，可以直接通过例子来理解：

```

1 std::vector v1 = { 1, 2 };
2 std::vector v2 = { 'a', 'b', 'c' };
3 std::vector v3 = { 3, 4, 5 };
4
5 fmt::print("zip: {}\n", std::views::zip(v1, v2));
6 fmt::print("zip_transform: {}\n", std::views::zip_transform(std::multiplies(), v1, v3));
7 fmt::print("adjacent: {}\n", v2 | std::views::pairwise());
8 fmt::print("adjacent_transofrm: {}\n", v3 | std::views::pairwise_transform(std::plus()));
9
10 // Output:
11 // zip: {(1, 'a'), (2, 'b')}
12 // zip_transform: {3, 8}
13 // adjacent: {('a', 'b'), ('b', 'c')}
14 // adjacent_transofrm: {7, 9}

```

`zip`可以将两个 Ranges 压缩为一个 Range（2 个的时候元素类型为 **pairs**，多个的时候元素类型为 **tuples**），`zip_transform`会在压缩的时候执行转换操作。

`ajacent`是一种特殊的`zip`，它的输入只有一个 Range，可以针对某几个元素进行分组，产生一个新的 Range，而`ajacent_transform`则会在分组之后执行转换操作。

其中，`pairwise`和`pairwise_transform`是`ajacent_view`和`ajacent_transform_view`按照两个单位分组时的别名。因此，也可以这样使用：

```

1 vector v = { 1, 2, 3, 4 };
2
3 for (auto i : v | views::adjacent<2>) {
4     // prints: (1, 2) (2, 3) (3, 4)
5     cout << '(' << i.first << ', ' << i.second << ") ";
6 }

```

13.30 Ranges: fold (P2322R6)

`fold`是数值算法`std::accumulate`更加通用的版本，包含`fold_left`和`fold_right`两个算法，用法如下：

```

1 // fold algorithms
2 int xs[] = { 1, 2, 3, 4, 5 };
3 auto concatl = [](std::string s, int i) { return s + std::to_string(i); };
4 auto concatr = [](int i, std::string s) { return s + std::to_string(i); };
5
6 auto fold_left = ranges::fold_left(xs, std::string(), concatl);
7 fmt::print("fold left: {}\n", fold_left);
8
9 auto fold_right = ranges::fold_right(xs, std::string(), concatr);
10 fmt::print("fold right: {}\n", fold_right);
11
12 // Output:
13 // fold left: 12345
14 // fold right: 54321

```

`std::accumulate`默认会执行累加操作，而`fold`表意更加广泛，所以加入了`<algorithm>`之中。

13.31 Ranges: iota, shift_left, and shift_right (P2440R1)

尽管 C++20 已经有了`iota_view`，但是针对已有 Ranges 来说还是不太方便。

线性初始化一个数组，C++23 之前有两种方法。

第一种方法是使用`std::iota`，该函数在 C++20 提供了`constexpr`版本，用法如下：

```

1 std::vector<int> vec(10);
2 std::iota(vec.begin(), vec.end(), 0);
3
4 fmt::print("{}\n", vec);
5
6 // Output:
7 // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

第二种方法是使用`std::views::iota`，用法如下：

```

1 std::vector<int> vec;
2 for (int i : std::views::iota(0, 10)) {
3     vec.push_back(i);
4 }
5
6 fmt::print("{}\n", vec);
7
8 // Output: Ditto

```

到了 C++23，算上`ranges::to`，又有了两种方法：

```

1 // 1. use ranges::to
2 std::vector<int> vec1 = views::iota(0, 10) | to<std::vector>();
3 fmt::print("{}\n", vec1);
4
5 // 2. use ranges::iota
6 std::vector<int> vec2(10);
7 ranges::iota(vec2, 0);
8 fmt::print("{}\n", vec2);

```

这些方法中，直接使用`ranges::iota`要更加便捷与高效。

`shift_left`与`shift_right`也是一组对称的算法，用法也比较简单，此处只展示一个：

```

1 std::vector v { 1, 2, 3, 4, 5 };
2 auto it = ranges::shift_left(v, 2);
3 // v = 3 4 5 4 5
4 //           ^
5 //           it
6 v.erase(it, v.end());
7
8 // [3 4 5]
9 fmt::print("{}\n", v);

```

注释很清晰，不多解释。

13.32 Range adaptors: slide, chunk, and chunk_by(P2442R1, P2443R1)

这是几个新的 Range adaptors，分别对应`slide_view`, `chunk_view`与`chunk_by_view`。先说前两个，简单的小例子：

```

1 std::vector v = {1, 2, 3, 4, 5};
2
3 // [[1, 2], [3, 4], [5]]
4 fmt::print("{}\n", v | std::views::chunk(2));
5
6 // [[1, 2], [2, 3], [3, 4], [4, 5]]
7 fmt::print("{}\n", v | std::views::slide(2));

```

意义很明显，`chunk`用于对数据进行「分块」操作，`slide`用于对数据进行连续「分块」操作，和`adjacent`比较相似。区别在于它的参数发生于运行期，而`adjacent`是模板参数，发生于编译期。

而`chunk_by`则可以根据某个条件进行「分块」，看个例子：

```

1 // chunk_by
2 std::vector v { 2, 1, 3, -4, 5 };

```

```

3
4 // [[2], [1, 3], [-4, 5]]
5 fmt::print("{}\n", v | views::chunk_by(std::less<>{}));

```

根据该 adaptor，便可以根据任意条件打散并重排数据。

13.33 Range adaptor: views::join_with (P2441R2)

这是一个与「分割」操作相反的「组合」操作 Range adaptor，对应[join_with_view](#)。用法其实很简单：

```

1 vector<string> vs = {"the", "quick", "brown", "fox"};
2 for (char c : vs | join_with('-')) {
3     cout << c;
4 }
5
6 // Output:
7 // the-quick-brown-fox

```

可以按照指定方式将所有元素组合起来。

13.34 std::generator (P2502R2)

这个是跟协程有关的，[std::generator](#)是一个 move-only 的 view，模拟了[input_range](#)。因此可以对其使用 Views，比如：

```

1 std::generator<int> ints(int start = 0) {
2     while (true)
3         co_yield start++;
4 }
5
6 void f() {
7     for (auto i : ints() | std::views::take(3))
8         std::cout << i << ' '; // prints 0 1 2
9 }

```

13.35 Fix istream_view (P2432R1)

[istream_view](#)在 C++20 存在一些基本的设计问题，在 23 进行了修复。

```

1 std::istringstream mystream { "0 1 2 3 4" };
2
3 // C++20 ERROR, C++23 OK
4 std::ranges::istream_view<int> v{ mystream };

```

13.36 ranges::contains (P2302R4)

在上一篇中介绍过ranges::find, ranges::search等等算法，要识别一个 Range 是否包含某个元素，或是否包含另一个子 Range，比较麻烦。

C++23 新引入了两个新算法：contains和contains_subrange，可以很好的满足该需求。

```
1 int arr1[] = { 4, 2, 3, 1 };
2 int arr2[] = { 4, 2 };
3 fmt::print("{}\n", contains(arr1, 4)); // true
4 fmt::print("{}\n", contains_subrange(arr1, arr2)); // true
```

Chapter 14

Three C++23 features for common use

👤 Marius Bancila 📅 2022-01-17 💬 ★★

C++23 is the current working version of the C++ standard. No major feature has been included so far, but a series of smaller ones as well as many defect reports have made it already to the standard. You can check the current status as well as the compiler support for the new features here. Many of these new features are small improvements or things you probably wouldn't use on a regular basis. However, I want to point here to three C++23 features that, in my opinion, stand out among the others as more likely to be used more often.

14.1 Literal suffixes for `size_t` and `ptrdiff_t`

`std::size_t` is an unsigned data type (of at least 16 bits) that can hold the maximum size of an object of any type. It can safely store the index of an array on any platform. It is the type returned by the `sizeof`, `sizeof...`, and `alignof` operators.

`std::ptrdiff_t` is a signed data type (of at least 17 bits) that represents the type of the result of subtracting two pointers.

In C++23, these have their own string literal suffixes.

Literal suffix	Deduced type	Example
uz or uZ or Uz or UZ	<code>std::size_t</code>	<code>auto a = 42uz;</code>
z or Z	<code>signed std::size_t</code> (<code>std::ptrdiff_t</code>)	<code>auto b = -42z;</code>

Let's see how this is useful. In C++20, we could write the following:

```
std::vector<int> v {1, 1, 2, 3, 5, 8};  
for(auto i = 0u; i < v.size(); ++i)  
{  
    std::cout << v[i] << '\n';  
}
```

The deduced type of the variable `i` is `unsigned int`. This works fine on 32-bit, where both `unsigned int` and `size_t`, which is the return type of the `size()` member function, are 32-bit. But on 64-bit you may get a warning and the value is truncated, because `unsigned int` is still 32-bit but `size_t` is 64-bit.

On the other hand, we can have the following:

```
std::vector<int> v {1, 1, 2, 3, 5, 8};
auto m = std::max(42, std::ssize(v)); // compiles on 32-bit but fails on 64-bit

std::vector<int> v {1, 1, 2, 3, 5, 8};
auto m = std::max(42ll, std::ssize(v)); // compiles on 64-bit but fails on 32-bit
```

Neither of these two versions work on both 32-bit and 64-bit platforms.

This is where the new literal suffixes help:

```
std::vector<int> v {1, 1, 2, 3, 5, 8};
for(auto i = 0uz; i < v.size(); ++i)
{
    std::cout << v[i] << '\n';
}
auto m = std::max(42z, std::ssize(v));
```

This code works the same on all platforms. See more¹².

14.2 Multidimensional subscript operator

Sometimes we need to work with multidimensional containers (or views). Accessing elements in an unidimensional container can be done with the subscript operator (such as `arr[0]` or `v[i]`). But for a multi-dimensional type, the subscript operator does not work nice. You cannot say `arr[0, 1, 2]`. The alternatives are:

- Define an access function, such as `at()` with any number of parameters (so you could say `c.at(0, 1, 2)`)
- overload the call operator (so you could say `c(0, 1, 2)`)
- overload the subscript operator with a brace-enclosed list (so you could say `c[{1,2,3}]`)
- chain single-argument array access operators (so you could say `c[0][1][2]`) which is probably leading to the least desirable APIs and usage

To demonstrate the point, let's consider a matrix class (that represents a two dimensional array). A simplistic implementation and usage is as follows:

```
1 template <typename T, size_t R, size_t C>
2 struct matrix
3 {
4     T& operator()(size_t const r, size_t const c) noexcept
5     {
6         return data_[r * C + c];
7     }
8
9     T const & operator()(size_t const r, size_t const c) const noexcept
```

¹<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0330r8.html>

²https://en.cppreference.com/w/cpp/language/integer_literal

```

10     {
11         return data_[r * C + c];
12     }
13
14     static constexpr size_t Rows = R;
15     static constexpr size_t Columns = C;
16 private:
17     std::array<T, R* C> data_;
18 };
19
20 int main()
21 {
22     matrix<int, 2, 3> m;
23     for (size_t i = 0; i < m.Rows; ++i)
24     {
25         for (size_t j = 0; j < m.Columns; ++j)
26         {
27             m(i, j) = i * m.Columns + (j + 1);
28         }
29     }
30
31     for (size_t i = 0; i < m.Rows; ++i)
32     {
33         for (size_t j = 0; j < m.Columns; ++j)
34         {
35             std::cout << m(i, j) << ' ';
36         }
37
38         std::cout << '\n';
39     }
40 }
```

I never liked the `m(i, j)` syntax, but this was the best we could do until C++23, IMO. Now, we can overload the subscript operator with multiple parameters:

```

1 T& operator[](size_t const r, size_t const c) noexcept
2 {
3     return data_[r * C + c];
4 }
5 T const & operator[](size_t const r, size_t const c) const noexcept
6 {
7     return data_[r * C + c];
8 }
```

We can now use the new `matrix` implementation as follows:

```

1 int main()
2 {
3     matrix<int, 3, 2> m;
4     for (size_t i = 0; i < m.Rows; ++i)
5     {
6         for (size_t j = 0; j < m.Columns; ++j)
7         {
8             m[i, j] = i * m.Columns + (j + 1);
9         }
10    }
11
12    for (size_t i = 0; i < m.Rows; ++i)
13    {
14        for (size_t j = 0; j < m.Columns; ++j)
15        {
16            std::cout << m[i, j] << ' ';
17        }
18
19        std::cout << '\n';
20    }
21 }
```

I just wished we had this twenty years ago! See also³⁴.

14.3 contains() member function for string/string_view

C++20 added the `starts_with()` and `ends_with()` member functions to `std::basic_string` and `std::basic_string_view`.

These enable us to check whether a string starts with a given prefix or ends with a given suffix.

```

1 int main()
2 {
3     std::string text = "lorem ipsum dolor sit amet";
4     std::cout << std::boolalpha;
5     std::cout << text.starts_with("lorem") << '\n'; // true
6     std::cout << text.starts_with("ipsum") << '\n'; // false
7     std::cout << text.ends_with("dolor") << '\n'; // false
8     std::cout << text.ends_with("amet") << '\n'; // true
9 }
```

Unfortunately, these don't help us checking whether a string contains a given substring. Of course, this is possible with the `find()` function. But this returns the position of the first character of the found substring or `npos` otherwise, so we need to do a check as follows:

³https://en.cppreference.com/w/cpp/language/operator_member_access

⁴<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2128r6.pdf>

```
std::cout << (text.find("dolor") != std::string::npos) << '\n';
```

I find this cumbersome and ugly when you just want to know if a string contains a particular substring or character.

In C++23, the circle is complete, as the same feature is available with the new `contains()` member function. This function enables us to check whether a substring or a single character is present anywhere the string. This is basically the same as `find(x) != npos`. But the syntax is nicer and in line with `starts_with()` and `ends_with()`.

```
std::cout << text.contains("dolor") << '\n';
```

See also⁵ ⁶ ⁷.

⁵<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1679r3.html>

⁶https://en.cppreference.com/w/cpp/string/basic_string/contains

⁷https://en.cppreference.com/w/cpp/string/basic_string_view/contains

Chapter 15

Three new utility functions in C++23

👤 Marius Bancila 📅 2022-11-08 💬 ★★

Some time ago I wrote a blog post called *Three C++23 features for common use*. In this article, I want to continue on that idea and discuss three new utility functions that were added to C++23.

15.1 std::unreachable

This new function is available in the `<utility>` header. It's intended to be used when you know you have an execution path in your code that cannot be reached but the compiler cannot figure that out. Informing the compiler could help to optimize impossible code branches.

A typical use case for this function are switch statements on a variable that can take only a limited set of values from its domain. For instance, an integer that can only be between 0 – 9. Here is a simple example with a switch that checks a `char` value and executes operations. Only a limited number of commands are supported but the argument is checked before invoking the function so it shouldn't be possible to receive other values than already handled in the switch.

```
1 void execute_command(char ch)
2 {
3     switch (ch)
4     {
5         case 'a':
6             /* add */ break;
7         case 'd':
8             /* delete */ break;
9         default:
10             std::unreachable();
11     }
12 }
13 bool is_valid(char ch)
14 {
15     return ch == 'a' || ch == 'd';
```

```

16 }
17 int main()
18 {
19     char ch = 0;
20     while (!is_valid(ch))
21         std::cin >> ch;
22     execute_command(ch);
23 }
```

It could be argued though that it might be safer to handle the default case in some way to ensure safe failure rather than assume that a variable/argument would always take a limited number of values. This depends on the particularities of your code.

It should be noted that `std::unreachable` invokes undefined behavior.

15.2 std::to_underlying

This is yet another utility function from the `<utility>` header. It converts an enumeration to its underlying type. It is basically syntactic sugar for the expression `static_cast<std::underlying_type_t<Enum> >(e)`.

Let's look at an example. Consider the following program:

```

1 void apply_style(int style)
2 {
3     std::cout << std::format("applying style {}\n", style);
4 }
5
6 enum class styles
7 {
8     A = 0x1,
9     B = 0x2,
10    C = 0x8000
11 };
12
13 int main()
14 {
15     apply_style(static_cast<int>(styles::C));
16 }
```

You need a `static_cast` to convert from the scoped enum `styles` to `int`. This can be simplified using `std::to_underlying`, as follows:

```

1 int main()
2 {
3     apply_style(std::to_underlying(styles::C));
4 }
```

The `static_cast` is actually moved to the body of this function, whose implementation can be as follows:

```

1 template <typename Enum>
2 constexpr auto to_underlying(Enum e) noexcept
3 {
4     return static_cast<std::underlying_type_t<Enum>>(e);
5 }
```

15.3 std::byteswap

This utility function is available in the `<bit>` header and reverses the bytes of an integral value. This is an addition to the bit utilities added with this header in C++20. Its purpose is to allow developers perform this byte reversal in a performant way without relying to compiler intrinsics.

Here is an example:

```

1 template <std::integral T>
2 void print_hex(T v)
3 {
4     for (std::size_t i = 0; i < sizeof(T); ++i, v >>= 8)
5     {
6         std::cout << std::format("{:02X} ", static_cast<unsigned>(T(0xFF) & v));
7     }
8     std::cout << '\n';
9 }
10
11 int main()
12 {
13     unsigned char a = 0xBA;
14     print_hex(a); // BA
15     print_hex(std::byteswap(a)); // BA
16     unsigned short b = 0xBAAD;
17     print_hex(b); // AD BA
18     print_hex(std::byteswap(b)); // BA AD
19     int c = 0xBAADF00D;
20     print_hex(c); // OD FO AD BA
21     print_hex(std::byteswap(c)); // BA AD FO OD
22     long long d = 0xBAADF00DBAADCOFE;
23     print_hex(d); // FE CO AD BA OD FO AD BA
24     print_hex(std::byteswap(d)); // BA AD FO OD BA AD CO FE
25 }
```

Byte swapping is important when transferring data between system that use different order for the sequence of bytes stores in memory. This is called endianness. In Big Endian systems, the most significant byte is stored at the smaller address (first). In Little Endian systems, the least significant byte is stored at

the smaller address. x86/x64 architectures use the little endian format. ARM supports both little and big endianness. Network protocols specify big endian for the order of transmission which requires swapping to/from systems using little endian order.

Chapter 16

C++23's Deducing this: what it is, why it is, how to use it

👤 Sy Brand 📅 2022-06-27 💬★★★

Deducing this (P0847) is a C++23 feature which gives a new way of specifying non-static member functions. Usually when we call an object's member function, the object is implicitly passed to the member function, despite not being present in the parameter list. P0847 allows us to make this parameter explicit, giving it a name and const/reference qualifiers. For example:

```
1 struct implicit_style {
2     void do_something(); //object is implicit
3 };
4
5 struct explicit_style {
6     void do_something(this explicit_style& self); //object is explicit
7 };
```

The explicit object parameter is distinguished by the keyword `this` placed before the type specifier, and is only valid for the first parameter of the function.

The reasons for allowing this may not seem immediately obvious, but a bunch of additional features fall out of this almost by magic. These include de-quadruplication of code, recursive lambdas, passing `this` by value, and a version of the CRTP¹ which doesn't require the base class to be templated on the derived class.

This post will walk through an overview of the design, then many of the cases you can use this feature for in your own code.

For the rest of this blog post I'll refer to the feature as "explicit object parameters", as it makes more sense as a feature name than `deducing this`. Explicit object parameters are supported in MSVC as of Visual Studio 2022 version 17.2. A good companion to this post is Ben Deane's talk *Deducing this Patterns*² from CppCon.

¹<https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/>

²<https://www.youtube.com/watch?v=jXf-bazhJw>

16.1 Overview

The paper which proposed this feature was written by Gašper Ažman, Ben Deane, Barry Revzin, and myself, and was guided by the experience of many experts in the field³. Barry and I began writing a version of this paper after we each implemented `std::optional` and came across the same problem. We would be writing the `value` function of `optional` and, like good library developers, we'd try to make it usable and performant in as many use-cases as we could. So we'd want `value` to return a `const` reference if the object it was called on was `const`, we'd want it to return an rvalue if the object it was called on was an rvalue, etc. It ended up looking like this:

```

1  template <typename T>
2  class optional {
3      // version of value for non-const lvalues
4      constexpr T& value() & {
5          if (has_value()) {
6              return this->m_value;
7          }
8          throw bad_optional_access();
9      }
10
11     // version of value for const lvalues
12     constexpr T const& value() const& {
13         if (has_value()) {
14             return this->m_value;
15         }
16         throw bad_optional_access();
17     }
18
19     // version of value for non-const rvalues... are you bored yet?
20     constexpr T&& value() && {
21         if (has_value()) {
22             return std::move(this->m_value);
23         }
24         throw bad_optional_access();
25     }
26
27     // you sure are by this point
28     constexpr T const&& value() const&& {
29         if (has_value()) {
30             return std::move(this->m_value);
31         }
32         throw bad_optional_access();

```

³<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0847r7.html#acknowledgements>

```

33     }
34     // ...
35 };

```

(If you’re not familiar with the `member_function_name() &` syntax, this is called “ref-qualifiers” and you can find more info on Andrzej Krzemieński’s blog⁴. If you’re not familiar with rvalue references (`T&&`) you can read up on move semantics on this Stack Overflow question⁵)

Note the near-identical implementations of four versions of the same function, only differentiated on whether they’re `const` and whether they move the stored value instead of copying it.

Barry and I would then move on to some other function and have to do the same thing. And again and again, over and over, duplicating code, making mistakes, building maintenance headaches for the future versions of ourselves. “What if”, we thought, “you could just write this?”

```

1 template <typename T>
2 struct optional {
3     // One version of value which works for everything
4     template <class Self>
5     constexpr auto&& value(this Self&& self) {
6         if (self.has_value()) {
7             return std::forward<Self>(self).m_value;
8         }
9         throw bad_optional_access();
10    }

```

(If you’re not familiar with `std::forward`, you can read about perfect forwarding on Eli Bendersky’s blog⁶)

This does the same thing as the above four overloads, but in a single function. Instead of writing different versions of `value` for `const optional&`, `const optional&&`, `optional&`, and `optional&&`, we write one function template which deduces the `const/volatile/reference` (cvref for short) qualifiers of the object the it is called on. Making this change for almost every function in the type would cut down our code by a huge amount.

So we wrote a version of what eventually got standardised, soon discovered that Gašper and Ben were working on a different paper for the exact same feature, we joined forces, and here we all are several years later.

16.2 Design

The key design principle we followed was that it should do what you expect. To achieve this, we touched as few places in the standard as we possibly could. Notably, we didn’t touch overload resolution rules or template deduction rules, and name resolution was only changed a little bit (as a treat).

As such, say we have a type like so:

```

1 struct cat {
2     template <class Self>

```

⁴<https://akrzemi1.wordpress.com/2014/06/02/ref-qualifiers/>

⁵<https://stackoverflow.com/questions/3106110/what-is-move-semantics>

⁶<https://eli.thegreenplace.net/2014/perfect-forwarding-and-universal-references-in-c>

```

3     void lick_paw(this Self&& self);
4 }

```

The template parameter `Self` will be deduced based on all of the same template deduction rules you’re already familiar with. There’s no additional magic. You don’t have to use the names `Self` and `self`, but I think they’re the clearest options, and this follows what several other programming languages do.

```

1 cat marshmallow;
2 marshmallow.lick_paw();                                //Self = cat&
3
4 const cat marshmallow_but_stubborn;
5 marshmallow_but_stubborn.lick_paw();                    //Self = const cat&
6
7 std::move(marshmallow).lick_paw();                      //Self = cat
8 std::move(marshmallow_but_stubborn).lick_paw();        //Self = const cat

```

One name resolution change is that inside such a member function, you are not allowed to explicitly or implicitly refer to `this`.

```

1 struct cat {
2     std::string name;
3
4     void print_name(this const cat& self) {
5         std::cout << name;           //invalid
6         std::cout << this->name; //also invalid
7         std::cout << self.name;   //all good
8     }
9 }

```

16.3 Use Cases

For the rest of this post, we’ll look at all the different uses of this feature (at least the ones discovered so far that I know of!) Many of these examples were taken straight from the paper.

16.3.1 De-duplication/quadruplication

We’ve already seen how the feature can be applied to a type such as `optional` to avoid having to write four overloads of the same function.

Note also that this lowers the burden on initial implementation and maintenance of dealing with rvalue member functions. Quite often developers will write only `const` and `non-const` overloads for member functions, since in many cases we don’t really want to write another two whole functions just to deal with rvalues. With deduced qualifiers on `this`, we get the rvalue versions for free: we just need to write `std::forward` in the right places to get the runtime performance gains which come with avoiding unnecessary copies:

```

1 class cat {
2     toy held_toy_;

```

```

3
4 public:
5     //Before explicit object parameters
6     toy& get_held_toy() { return held_toy_; }
7     const toy& get_held_toy() const { return held_toy_; }

8
9     //After
10    template <class Self>
11        auto&& get_held_toy(this Self&& self) {
12            return self.held_toy_;
13        }

14
15    //After + forwarding
16    template <class Self>
17        auto&& get_held_toy(this Self&& self) {
18            return std::forward<Self>(self).held_toy_;
19        }
20    };

```

Of course for a simple getter like this, whether or not this change is worth it for your specific use case is up to you. But for more complex functions, or cases where you are dealing with large objects which you want to avoid copying, explicit object parameters make this much easier to handle.

16.3.2 CRTP

The Curiously Recurring Template Pattern (CRTP) is a form of compile-time polymorphism which allows you to extend types with common pieces of functionality without paying the runtime costs of virtual functions. This is sometimes referred to as mixins (this isn’t all the CRTP can be used for, but it is the most common use). For example, we could write a type `add_postfix_increment` which can be mixed in to another type in order to define postfix increment in terms of prefix increment:

```

1 template <typename Derived>
2 struct add_postfix_increment {
3     Derived operator++(int) {
4         auto& self = static_cast<Derived&>(*this);
5
6         Derived tmp(self);
7         ++self;
8         return tmp;
9     }
10 };
11
12 struct some_type : add_postfix_increment<some_type> {
13     // Prefix increment, which the postfix one is implemented in terms of

```

```

14     some_type& operator++();
15 };

```

Templating a base class on its derived cast and `static_casting this` inside the function can be a bit arcane, and the problem gets worse when you have multiple levels of CRTP. With explicit object parameters, since we didn't change template deduction rules, the type of the explicit object parameter can be deduced to a derived type. More concretely:

```

1 struct base {
2     template <class Self>
3     void f(this Self&& self);
4 };
5
6 struct derived : base {};
7
8 int main() {
9     derived my_derived;
10    my_derived.f();
11 }

```

In the call `my_derived.f()`, the type of `Self` inside `f` is `derived&`, not `base&`.

This means that we can define the above CRTP example like so:

```

1 struct add_postfix_increment {
2     template <typename Self>
3     auto operator++(this Self&& self, int) {
4         auto tmp = self;
5         ++self;
6         return tmp;
7     }
8 };
9
10 struct some_type : add_postfix_increment {
11     // Prefix increment, which the postfix one is implemented in terms of
12     some_type& operator++();
13 };

```

Note that now `add_postfix_increment` is not a template. Instead, we've moved the customisation to the postfix `operator++`. This means we don't need to pass `some_type` as a template argument anywhere: everything "just works".

16.3.3 Forwarding out of lambdas

Copying captured values out of a closure is simple: we can just pass around the object as usual. Moving captured values out of a closure is also simple: we can just call `std::move` on it. A problem occurs when we need to perfect-forward a captured value based on whether the closure is an lvalue or rvalue.

One use case I stole from P2445 is for lambdas which can be used in both “retry” and “try or fail” contexts:

```
auto callback = [m=get_message(), &scheduler]() -> bool {
    return scheduler.submit(m);
};

callback(); // retry(callback)
std::move(callback()); // try-or-fail(rvalue)
```

The question here is: how do we forward `m` based on the value category of the closure? Explicit object parameters give us the answer. Since a lambda generates a class with an `operator()` member function of the given signature, all the machinery I’ve just explained works for lambdas too.

```
auto closure = [](this auto&& self) {
    //can use self inside the lambda
};
```

This means we can perfect-forward based on the value category of the closure inside the lambda. P2445 gives a `std::forward_like` helper, which forwards some expression based on the value category of another:

```
auto callback = [m=get_message(), &scheduler](this auto &&self) -> bool {
    return scheduler.submit(std::forward_like<decltype(self)>(m));
};
```

Now our original use case works, and the captured object will be copied or moved depending on how we use the closure.

16.3.4 Recursive lambdas

Since we now have the ability to name the closure object in a lambda’s parameter list, this allows us to do recursive lambdas! As above:

```
auto closure = [](this auto&& self) {
    self(); //just call ourself until the stack overflows
};
```

There are more useful uses for this than just overflowing stacks, though. Consider, for example, the ability to do visitation of recursive data structures without having to define additional types or functions? Given the following definition of a binary tree:

```
struct Leaf { };
struct Node;
using Tree = std::variant<Leaf, Node*>;
struct Node {
    Tree left;
    Tree right;
};
```

We can count the number of leaves like so:

```

1 int num_leaves(Tree const& tree) {
2     return std::visit(overload( //see below
3         [](Leaf const&) { return 1; },
4         [](this auto const& self, Node* n) -> int {
5             return std::visit(self, n->left) + std::visit(self, n->right);
6         }
7     ), tree);
8 }
```

`overload` here is some facility to create an overload set from multiple lambdas, and is commonly used for `variant` visitation. See [cppreference⁷](#), for example.

This counts the number of leaves in the tree through recursion. For each function call in the call graph, if the current is a `Leaf`, it returns `1`. Otherwise, the overloaded closure calls itself through `self` and recurses, adding together the leaf counts for the left and right subtrees.

16.3.5 Pass this by value

Since we can define the qualifiers of the now-explicit object parameter, we can choose to take it by value rather than by reference. For small objects, this can give us better runtime performance. In case you're not familiar with how this affects code generation, here's an example.

Say we have this code, using regular old implicit object parameters:

```

1 struct just_a_little_guy {
2     int how_smol;
3     int uwu();
4 };
5
6 int main() {
7     just_a_little_guy tiny_tim{42};
8     return tiny_tim.uwu();
9 }
```

MSVC generates the following assembly:

```

1 sub    rsp, 40
2 lea    rcx, QWORD PTR tiny_tim$[rsp]
3 mov    DWORD PTR tiny_tim$[rsp], 42
4 call   int just_a_little_guy::uwu(void)
5 add    rsp, 40
6 ret    0
```

I'll walk through this line-by-line.

- `sub rsp, 40` allocates 40 bytes on the stack. This is 4 bytes to hold the `int` member of `tiny_tim`, 32 bytes of *shadow space*⁸ for `uwu` to use, and 4 bytes of padding.

⁷<https://en.cppreference.com/w/cpp/utility/variant/visit>

⁸<https://stackoverflow.com/questions/30190132/what-is-the-shadow-space-in-x64-assembly>

- The `lea` instruction loads the address of the `tiny_tim` variable into the `rcx` register, which is where `uwu` is expecting the implicit object parameter (due to the *calling conventions*⁹ used).
- The `mov` stores `42` into the int member of `tiny_tim`.
- We then call the `uwu` function.
- Finally we de-allocate the space we allocated on the stack before and return.

What happens if we instead specify `uwu` to take its object parameter by value, like this?

```

1 struct just_a_little_guy {
2     int how_smol;
3     int uwu(this just_a_little_guy);
4 };

```

In that case, the following code is generated:

```

1 mov    ecx, 42
2 jmp    static int just_a_little_guy::uwu(this just_a_little_guy)

```

We just move `42` into the relevant register and jump (`jmp`) to the `uwu` function. Since we're not passing by-reference we don't need to allocate anything on the stack. Since we're not allocating on the stack we don't need to de-allocate at the end of the function. Since we don't need to deallocate at the end of the function we can just jump straight to `uwu` rather than jumping there and then back into this function when it returns, using `call`.

These are the kinds of optimisations which can prevent “death by a thousand cuts” where you take small performance hits over and over and over, resulting in slower runtimes that are hard to find the root cause of.

16.3.6 SFINAE-unfriendly callables

This issue is a bit more esoteric, but does actually happen in real code (I know because I got a bug report on my extended implementation of `std::optional` which hit this exact issue in production). Given a member function of `optional` called `transform`, which calls the given function on the stored value only if there is one, the problem looks like this:

```

1 struct oh_no {
2     void non_const();
3 };
4
5 tl::optional<oh_no> o;
6 o.transform([](auto&& x) { x.non_const(); }); //does not compile

```

The error which MSVC gives for this looks like:

```
error C2662: 'void oh_no::non_const(void)': cannot convert 'this' pointer
from 'const oh_no' to 'oh_no &'
```

⁹<https://docs.microsoft.com/cpp/build/x64-calling-convention?view=msvc-170>

So it's trying to pass a `const oh_no` as the implicit object parameter to `non_const`, which doesn't work. But where did that `const oh_no` come from? The answer is inside the implementation of `optional` itself. Here is a deliberately stripped-down version:

```

1 template <class T>
2 struct optional {
3     T t;
4
5     template <class F>
6     auto transform(F&& f) -> std::invoke_result_t<F&&, T&>;
7
8     template <class F>
9     auto transform(F&& f) const -> std::invoke_result_t<F&&, const T&&>;
10 };

```

Those `std::invoke_result_t`s are there to make `transform` SFINAE-friendly¹⁰. This basically means that you can check whether a call to `transform` would compile and, if it wouldn't, do something else instead of just aborting the entire compilation. However, there's a bit of a hole in the language here.

When doing overload resolution on `transform`, the compiler has to work out which of those two overloads is the best match given the types of the arguments. In order to do so, it has to instantiate the declarations of both the `const` and `non-const` overloads. If you pass an invocable to `transform` which is not itself SFINAE-friendly, and isn't valid for a `const` qualified implicit object (which is the case with my example) then instantiating the declaration of the `const` member function will be a hard compiler error. Oof.

Explicit object parameters allow you to solve this problem because the cvref qualifiers are deduced from the expression you call the member function on: if you never call the function on a `const optional` then the compiler never has to try and instantiate that declaration. Given `std::copy_cvref_t` from P1450:

```

1 template <class T>
2 struct optional {
3     T t;
4
5     template <class Self, class F>
6     auto transform(this Self&& self, F&& f)
7         -> std::invoke_result_t<F&&, std::copy_cvref_t<Self, T>>;
8 };

```

This allows the above example to compile while still allowing `transform` to be SFINAE-friendly.

16.4 Conclusion

I hope this has helped clarify the function and utility of explicit object parameters. You can try out the feature in Visual Studio version 17.2. If you have any questions, comments, or issues with the feature, you can comment below, or reach us via email at visualcpp@microsoft.com or via Twitter at @VisualC.

¹⁰<https://stackoverflow.com/questions/35033306/what-does-it-mean-when-one-says-something-is-sfinae-friendly>

Chapter 17

Using the C++23 std::expected type

👤 Marius Bancila 📅 2022-08-17 💬 ★★★

The C++23 standard will feature a new utility type called `std::expected`. This type either contains an expected value, or an unexpected one, typically providing information about the reason something failed (and the expected value could not be returned). This feature is, at this time, supported in GCC 12 and MSVC 19.33 (Visual Studio 2022 17.3). In this article, we'll see what `std::expected` is and how it can be used.

17.1 Why do we need std::expected?

Suppose you have to write a function that returns some data. It has to perform one or more operations that may fail. This function needs to return the data, but also needs to indicate failure and the cause for the failure. There are different ways to implement this.

17.1.1 Alternative 1: status code + reference parameter

One alternative is to return a status code indicating success or the reason of failure. Additionally, the actual returned data is a parameter passed by reference.

```
1 enum class Status
2 {
3     Ok,
4     AccessDenied,
5     DataSourceError,
6     DataError,
7 };
8
9 bool HasAccess() { return true; }
10 int OpenConnection() { return 0; }
11 int Fetch() { return 0; }
12
13 Status ReadData(std::vector<int>& data)
14 {
```

```

15     if (!HasAccess())
16         return Status::AccessDenied;
17     if (OpenConnection() != 0)
18         return Status::DataSourceError;
19     if (Fetch() != 0)
20         return Status::DataError;
21     data.push_back(42);
22     return Status::Ok;
23 }
```

This is how it can be used:

```

1 void print_value(int const v)
2 {
3     std::cout << v << '\n';
4 }
5
6 int main()
7 {
8     std::vector<int> data;
9     Status result = ReadData(data);
10    if (result == Status::Ok)
11    {
12        std::ranges::for_each(data, print_value);
13    }
14    else
15    {
16        std::cout << std::format("Error code: {}\\n", (int)result);
17    }
18 }
```

17.1.2 Alternative 2: using exceptions

Another alternative is to return the actual data but in case of failure throw an exception.

```

1 struct status_exception : public std::exception
2 {
3     status_exception(Status status) : std::exception(), status_(status) {}
4     status_exception(Status status, char const* const message)
5         : std::exception(message), status_(status) {}
6     Status status() const { return status_; }
7 private:
8     Status status_;
9 };
```

10

```

11 std::vector<int> ReadData()
12 {
13     if (!HasAccess())
14         throw status_exception(Status::AccessDenied);
15     if (OpenConnection() != 0)
16         throw status_exception(Status::DataSourceError);
17     if (Fetch() != 0)
18         throw status_exception(Status::DataError);
19     std::vector<int> data;
20     data.push_back(42);
21     return data;
22 }
```

This time, we need to try-catch the call:

```

1 int main()
2 {
3     try
4     {
5         auto data = ReadData();
6         std::ranges::for_each(data, print_value);
7     }
8     catch (status_exception const& e)
9     {
10        std::cout << std::format("Error code: {}\n", (int)e.status());
11    }
12 }
```

Choosing between one of these could be a personal choice or may depend on imposed restrictions. For instance, there could be a no-exceptions requirement, in which case the 2nd alternative cannot be used.

17.1.3 Alternative 3: using std::variant

Another possible options, in C++17, is to use `std::variant`. In this case, our function could looks as follows:

```

1 std::variant<std::vector<int>, Status> ReadData()
2 {
3     if (!HasAccess())
4         return { Status::AccessDenied };
5     if (OpenConnection() != 0)
6         return { Status::DataSourceError };
7     if (Fetch() != 0)
8         return { Status::DataError };
9     std::vector<int> data;
10    data.push_back(42);
```

```

11     return data;
12 }
```

However, when it comes to using it, it gets nasty. We need to visit each possible alternative of the variant type and the syntax to do so is horrendous.

```

1 int main()
2 {
3     auto result = ReadData();
4     std::visit([](auto& arg) {
5         using T = std::decay_t<decltype(arg)>;
6         if constexpr (std::is_same_v<T, std::vector<int>>)
7         {
8             std::ranges::for_each(arg, print_value);
9         }
10        else if constexpr (std::is_same_v<T, Status>)
11        {
12             std::cout << std::format("Error code: {}\n", (int)arg);
13         }
14     }, result);
15 }
```

In my opinion, `std::variant` is difficult to use, and I don't like making use of it.

Note: you can read more about `std::variant` in this article: *std::visit is everything wrong with modern C++¹*.

17.1.4 Alternative 4: using std::optional

The `std::optional` type may contain or may not contain a value. This can be used when returning no data is a valid option for a function that normally would return a value. Like in our case:

```

1 std::optional<std::vector<int>> ReadData()
2 {
3     if (!HasAccess()) return {};
4     if (OpenConnection() != 0) return {};
5     if (Fetch() != 0) return {};
6     std::vector<int> data;
7     data.push_back(42);
8     return data;
9 }
```

We can use this as follows:

```

1 int main()
2 {
```

¹<https://bitashing.io/std-visit.html>

```

3     auto result = ReadData();
4     if (result)
5     {
6         std::ranges::for_each(result.value(), print_value);
7     }
8     else
9     {
10        std::cout << "No data\n";
11    }
12 }
```

The `std::optional` type has several members for checking and accessing the value, including:

- `has_value()` (e.g. `if(result.has_value())`) checks whether the object contains a value
- `operator bool` (e.g. `if(result)`) performs the same check
- `value()` (e.g. `result.value()`) returns the contained value or throws `std::bad_optional_access` if the object does not contain a value
- `value_or()` (e.g. `result.value_or(...)`) returns the contained value or the supplied one if the object does not contain any value
- `operator->` and `operator*` (e.g. `*result`) access the contained value but have undefined behavior if the object does not contain any value

The problem with this particular implementation of `ReadData` is that we didn't get the reason for the failure back. To do so, we would either need to introduce a function parameter (passed by reference) or throw an exception (like with the second alternative presented earlier).

17.2 Enter `std::expected`

In C++23, we get this new utility type, `std::expected<T, E>`, in the new `<expected>` header. This is supposed to be used for functions that return a value but may encounter some errors in which case they may return something else, such as information about the error. In a way, `std::expected` is a combination of `std::variant` and `std::optional`. On one hand, it's a discriminated union, it either hold a `T` (the expected type) or an `E` (the unexpected type). This is at least, logically; but more of this, shortly. On the other hand, it was an interface similar to `std::optional<T>`:

- `has_value()` (e.g. `if(result.has_value())`) returns true if the object contains the expected value (not the unexpected one)
- `operator bool` (e.g. `if(result)`) same as `has_value`
- `value()` (e.g. `result.value()`) returns the expected value if the object contains one or throws `std::bad_expected_access<E>`, an exception type that contains the unexpected value stored by the `std::expected<T, E>` object

- `value_or()` (e.g. `result.value_or(...)`) returns the expected value if the object contains one or, otherwise, the supplied value
- `error()` returns the unexpected value contained by the `std::expected<T, E>` object
- `operator->` and `operator*` access the expected value, if the object contains one; otherwise, the behavior is undefined

Let's see how the `ReadData` function may look when using `std::expected<T, E>` for the return type:

```

1 std::expected<std::vector<int>, Status> ReadData()
2 {
3     if (!HasAccess())
4         return std::unexpected<Status> { Status::AccessDenied };
5     if (OpenConnection() != 0)
6         return std::unexpected<Status> { Status::DataSourceError };
7     if (Fetch() != 0)
8         return std::unexpected<Status> { Status::DataError };
9     std::vector<int> data;
10    data.push_back(42);
11    return data;
12 }
```

This implementation can be used as follows:

```

1 int main()
2 {
3     auto result = ReadData();
4     if (result)
5     {
6         std::ranges::for_each(result.value(), print_value);
7     }
8     else
9     {
10        std::cout << std::format("Error code: {}\n", (int)result.error());
11    }
12 }
```

In this implementation, when an error occurs, an `std::unexpected<Status>` value is returned. This `std::unexpected` is a class template that acts as a container for an unexpected value of type `E`. The `std::expected<T, E>` models a discriminated union of types `T` and `std::unexpected<E>`.

In the previous example, the different functions called by `ReadData` had different ways of indicating success (and returning data). When you have an algorithm, or routine that is made of smaller parts, and each part is a function that returns the same `std::expected` instantiation, the calls could be easily chained. Here is an example. Let's consider a function that builds a user's avatar, adding a frame, badge, and text to an existing image. For this, let's assume the following stubs:

```

1  struct avatar {};
2  enum class error_code
3  {
4      ok,
5      error,
6  };
7
8  using avatar_result = std::expected<avatar, error_code>;
9  avatar_result add_frame(avatar const& a) {
10     return a; /* std::unexpected<error_code>(error_code::error); */
11 }
12 avatar_result add_badge(avatar const& a) {
13     return a; /* std::unexpected<error_code>(error_code::error); */
14 }
15 avatar_result add_text(avatar const& a) {
16     return a; /* std::unexpected<error_code>(error_code::error); */
17 }
```

Using these, we can write the following `make_avatar` function:

```

1  avatar_result make_avatar(avatar const& a,
2      bool const with_frame, bool const with_badge, bool const with_text)
3  {
4      avatar_result result = a;
5      if (with_frame)
6      {
7          result = add_frame(*result);
8          if (!result)
9              return result;
10     }
11     if (with_badge)
12     {
13         result = add_badge(*result);
14         if (!result)
15             return result;
16     }
17     if (with_text)
18     {
19         result = add_text(*result);
20         if (!result)
21             return result;
22     }
23     return result;
24 }
```

Each step is handled in the same manner and the code is very simple. This `make_avatar` function can be used as follows:

```
1 int main()
2 {
3     avatar a;
4     auto result = make_avatar(a, true, true, false);
5
6     if (result)
7     {
8         std::cout << "success\n";
9     }
10    else
11    {
12        std::cout << "Error: " << (int)result.error() << '\n';
13    }
14 }
```

Chapter 18

C++23: Consteval if to make compile time programming easier

👤 Sandor Dargo 📅 2022-06-01 🎉 ★

Let's continue our exploration of C++23 features! This week we discuss *how to call consteval functions from not explicitly constant evaluated ones*¹.

This paper², this new feature is also a good example to see how C++ evolves. C++20 introduced 2 new keywords, `consteval` and `constinit`. Although they've been good additions, in the meanwhile the community found some bugs and also came up with some ideas for improvement. And here they are shipped with the next version of C++!

18.1 What is if consteval?

The amount of `const*` syntax is clearly growing in C++. `const` was part of the original language and then we got `constexpr` with C++11. C++17 introduced `if constexpr`, C++20 brought us `consteval` and `constinit`, and with C++23 we are going to get `if consteval` (often referred to as consteval if).

Let's see what the latest addition is about.

A `consteval if` statement has no condition. Better to say, it's the condition itself. If it's evaluated in a manifestly constant-evaluated context, then the following compound statement is executed. Otherwise, it's not. In case there is an `else` branch present, it will be executed as you'd expect it.

If that helps readability, you can also use `if !consteval`. The following two pieces of code are equivalent.

```
1 if !consteval {  
2     foo();  
3 } else {  
4     bar();  
5 }  
6
```

¹https://en.cppreference.com/w/cpp/language/if/#Consteval_if

²<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1938r0.html>

```

7 // same as
8
9 if consteval {
10     bar();
11 } else {
12     foo();
13 }
```

18.2 How to call consteval functions?

To answer that question, let's remind us of the difference between a `constexpr` and a `consteval` function. A `constexpr` function's return value can be computed at compile-time or during run-time. A `consteval` function is guaranteed to be executed during compile time, it's also called an **immediate function**.

In C++, we have a tendency of moving more and more computations to compile time. As such we slightly increase the compile-time (although it still goes down due to better compilers and more powerful computers), but we speed up the runtime. Following these trends and benefitting from compile-time computations, you might want to call `consteval` functions from `constexpr` functions. But it's not going to work with C++20.

```

1 consteval int bar(int i) {
2     return 2*i;
3 }
4
5 constexpr int foo(int i) {
6     return bar(i);
7 }
8
9 int main() {
10    [[maybe_unused]] auto a = foo(5);
11 }
12 /*
13 In function 'constexpr int foo(int)':
14 error: 'i' is not a constant expression
15     |      return bar(i);
16     |      ~~~^~~
17 */
```

It makes sense. After all, as `foo(int)` is a `constexpr` function, it can be executed at runtime too. But what if you really want to call a `consteval` function from a `constexpr` function when it's executed at compile time?

In C++20, `consteval` functions could invoke `constexpr` ones, but not the other way around. Even if you try to surround the call of the `consteval` function with `std::is_constant_evaluated()`, it won't change. The following example is not going to work, because `i` is still not a constant expression:

```

1 consteval int bar(int i) {
2     return 2*i;
3 }
4
5 constexpr int foo(int i) {
6     if (std::is_constant_evaluated()) {
7         return bar(i);
8     }
9     return 2*i;
10 }
11
12 int main() {
13     [[maybe_unused]] auto a = foo(5);
14 }
15 /*
16 main.cpp: In function 'constexpr int foo(int)':
17 main.cpp:6:14: error: 'is_constant_evaluated' is not a member of 'std'
18     6 |     if (std::is_constant_evaluated()) {
19     |             ^
20 main.cpp:7:19: error: 'i' is not a constant expression
21     7 |         return bar(i);
22     |             ~~~^~~
23
24 */

```

This proposal³ fixes it, by adding the new language feature of `if consteval`. Use that to call consteval functions from constexpr ones. In fact, not only from constexpr ones but from any function. Just make sure that you set the `-std=c++2b` compiler flag.

```

1 consteval int bar(int i) {
2     return 2*i;
3 }
4
5 int foo(int i) {
6     if consteval {
7         return bar(i);
8     }
9     return 2*i;
10 }
11
12 int main() {
13     [[maybe_unused]] auto a = foo(5);
14 }

```

³<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1938r0.html>

While `if consteval` behaves exactly as `if (std::is_constant_evaluated)`, it's superior to it because it doesn't need any header include, its syntax is crystal clear, plus you can invoke `consteval` functions if it evaluates to true.

18.3 Conclusion

In this article, we learnt about a new C++ feature, `if consteval` that will help us invoke `consteval` functions when the context is constant-evaluated, yet it's not explicitly declared so.

Chapter 19

C++23: Narrowing contextual conversions to bool

▀ Sandor Dargo 2022-06-15 🔖 ★

In the previous article discussing new language features of C++23, we discussed `if consteval`. Today, we'll slightly discuss `if constexpr` and also `static_assert`. Andrzej Krzemieński proposed a paper¹ to make life a bit easier by allowing a bit more implicit conversions. Allowing a bit more narrowing in some special contexts.

19.1 A quick recap

For someone less experienced with C++, let's start with recapitulating what the most important concepts of this paper represent.

19.1.1 `static_assert`

Something I just learned is that `static_assert` was introduced by C++11. I personally thought that it was a much older feature. It serves for performing compile-time assertions. It takes two parameters

- a boolean constant expression
- a message to be printed by the compiler in case of the boolean expression is `false`. C++17 made this message optional.

With `static_assert` we can assert the characteristics of types at compiler time (or anything else that is available knowledge at compile time).

```
1 #include <type_traits>
2
3 class A {
4 public:
5 // uncomment the following line to break the first assertion
```

¹里缪注：指的是 P1401, <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1401r5.html>

```

6 // virtual ~A() = default;
7 };
8
9 int main() {
10     static_assert(std::is_trivial_v<A>, "A is not a trivial type");
11     static_assert(1 + 1 == 2);
12 }
```

19.1.2 constexpr if

`if constexpr` is a feature introduced in C++17. Based on a constant expression condition, with **constexpr if** we can select and discard which branch to compile.

Take the following example from C++ Reference²:

```

1 template<typename T>
2 auto get_value(T t)
3 {
4     if constexpr (std::is_pointer_v<T>)
5         return *t; // deduces return type to int for T = int*
6     else
7         return t; // deduces return type to int for T = int
8 }
```

If `T` is a pointer, then the `template` will be instantiated with the first branch and the `else` part be ignored. In case, it's a value, the `if` part will be ignored and the `else` is kept. `if constexpr` has been a great addition that helps us simplify SFINAE and whatever code that is using `std::enable_if`.

19.2 Narrowing

Narrowing is a type of conversion. When that happens the converted value is losing from its precision. Most often it's something to avoid, just like *the Core Guidelines says in ES.46*³.

Converting a `double` to an `int`, a `long` to an `int`, etc., are all narrowing conversions where you (potentially) lose data. In the first case, you lose the fractions and in the second, you might already store a number that is bigger than the target type can store.

Why would anyone want that implicitly?

But converting an `int` to a `bool` is also narrowing and that can be useful. When that happens `0` is converted to `false`, and anything else (including negative numbers) will result in `true`.

Let's see how the paper wants to change the status quo.

19.3 What is the paper proposing

In fact, the proposal of Andrzej might or might not change anything for you depending on your compiler and its version. On the other hand, it definitely makes the standard compiler compliant.

²<https://en.cppreference.com/w/cpp/language/if>

³<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-narrowing>

Wait, what?

Let's take the following piece of code.

```

1 template <std::size_t N>
2 class Array
3 {
4     static_assert(N, "no 0-size Arrays");
5     // ...
6 };
7
8 Array<16> a;
```

According to - the pre-paper-acceptance version of - the standard, it should fail to compile because **N** which is 16 shouldn't be narrowed to **bool**. Still, if you compile the code with the major implementations, it will compile without any issue.

The paper updates the standard so that it matches this behaviour.

The other context where the paper changes the standard is **if constexpr**. Converting contextually a type to bool is especially useful with enums used as flags. Let's have a look at the following example:

```

1 enum Flags { Write = 1, Read = 2, Exec = 4 };
2
3 template <Flags flags>
4 int f() {
5     if constexpr (flags & Flags::Exec) // should fail to compile due to narrowing
6         return 0;
7     else
8         return 1;
9 }
10
11 int main() {
12     return f<Flags::Exec>(); // when instantiated like this
13 }
```

As the output of **flags & Flags::Exec** is an **int**, according to the standard it shouldn't be narrowed down to a **bool**, while the intentions of the coder are evident.

Earlier versions of Clang failed to compile this piece of code, you had to cast the condition to **bool** explicitly. Yet, later versions and all the other major compilers compiled the successfully.

There are 2 other cases where the standard speaks about “*contextually converted constant expression of type **bool***”, but the paper doesn't change the situation for those. For more details on that, check out the paper!

19.4 Conclusion

P1401R5 will not change how we code, it will not or just slightly change how compilers work. But it makes the major compilers compliant with the standard. It aligns the standard with the implemented

behaviour and will officially let the compilers perform a narrowing conversion to `bool` in the contexts of a `static_assert` or `if constexpr`. Now we can avoid explicitly casting expressions to `bool` in these contexts without guilt. Thank you, Andrzej!

Chapter 20

6 C++23 features improving string and string_view

• Sandor Dargo  2022-07-20  ★★

In this blog post, let's collect a couple of changes that are going to be shipped with C++23 and are all related to `strings` or `string_views`.

20.1 std::string and std::string_view have contains

One of C++20's useful addition to maps were the `contains` member function. We could replace the cumbersome to read query of `myMap.find(key) != myMap.end()` with the very easy to understand `myMap.contains(key)`. With C++23, `std::string` and `std::string_view` will have similar capabilities. You can call `contains()` with either a string or a character and it will return `true` or `false` depending on whether the queried `string` or `string_view` contains the input parameter.

```
1 #include <iostream>
2 #include <string>
3 #include <iomanip>
4
5 int main() {
6     std::string s{"there is a needle in the haystack"};
7     std::string_view sv{"acdef"};
8
9     if (s.contains("needle")) {
10         std::cout << "we found a needle in: " << std::quoted(s) << '\n';
11     }
12
13     if (!sv.contains('b')) {
14         std::cout << "we did not find a 'b' in: " << std::quoted(sv) << '\n';
15     }
16 }
```

```

17  /*
18  we found a needle in: "there is a needle in the haystack"
19  we did not find a 'b' in: "acdef"
20  */

```

20.2 No more undefined behaviour due to construction from nullptr

In an earlier newsletter, we discussed that initializing a `string` from a `nullptr` is undefined behaviour. In practice, this might happen when you convert a `const char *` to a `string`. What happens then? It depends on the compiler, `gcc` for example, throws a runtime exception.

Thanks to P2166R1¹, this is not something to worry about.

Instead of undefined behaviour, the constructor and assignment operator overloaded with `nullptr_t` are deleted and therefore compilation fails when you attempt to construct a new `string` out of a `nullptr`.

```

1 std::string s(nullptr);
2 /*
3 <source>:18:26: error: use of deleted function 'std::__cxx11::basic_string
4     <_CharT, _Traits, _Alloc>::basic_string(std::nullptr_t)
5     [with _CharT = char; _Traits = std::char_traits<char>;
6     _Alloc = std::allocator<char>; std::nullptr_t = std::nullptr_t] '
7     18 |     std::string s(nullptr);
8     |
9 /opt/compiler-explorer/gcc-12.1.0/include/c++/12.1.0/bits/basic_string.h:734:7:
10 note: declared here
11 734 |     basic_string(nullptr_t) = delete;
12     |     ^~~~~~
13 */

```

While this change is good and points in a good direction, not all of our problems disappear with `nullptrs`. Taking a `nullptr` and a size in the constructor (e.g. `std::string s(nullptr, 3)`) is still valid and remains undefined behaviour.

These changes are also valid for `string_view`.

20.3 Build `std::string_view` from ranges

With C++23, our favourite `string_view` doesn't only loses a constructor (the overload with a `nullptr` gets deleted), but also receives a new one. Soon, we'll be able to construct one out of a range directly.

So far, if we wanted to create a `string_view` out of a “range”, we had to invoke the constructor with a `begin` and an `end` iterators: `std::string_view sv(myRange.begin(), myRange.end())`. Now we'll be able to directly construct a `string_view` based on a range: `std::string_view sv(myRange)`.

¹<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2166r1.html>

20.4 basic_string::resize_and_overwrite()

One of the main reasons to use C++ is its high performance. An area where we often use the language in a non-efficient way is string handling. C++23 will bring us another string member function² that will help us to handle strings in a more performant way.

`std::string::resize_and_overwrite()` takes two parameters, a count and an operation and does the following (while returns nothing):

- if the count is smaller or equal to the `size()` of the string, it erases the last `size() - count` elements
- if `count` is larger than `size()`, appends `n - size()`³ default-initialized elements
- it also invokes `erase(begin() + op(data(), count), end())`.

In other words, `resize_and_overwrite()` will make sure that the given string has continuous storage containing `count + 1` characters.

If `op()` throws, the behaviour is undefined. It's also undefined if it tries to modify `count`.

But what can be an operation?

An operation is a function or function object to set the new contents of the string and it takes two parameters. The first one is the pointer to the first character in the string's storage and the second one is the same as `count`, the maximal possible new size of the string. It should return the actual new length of the string.

You have to pay attention that this operation doesn't modify the maximum size, does not try to set a longer string and doesn't modify the address of the first character either. That would mean undefined behaviour.

If correctly used, it'll help add some new content or rewrite the existing one. Or you can actually remove content. To illustrate this latter example, let's have a look at the second example of the original documentation.

```

1 std::string s { "Food: " };
2 s.resize_and_overwrite(10, [](char* buf, int n) {
3     return std::find(buf, buf + n, ':') - buf;
4 });
5 std::cout << "2. " << std::quoted(s) << '\n';
6 // 2. "Food"

```

Even though `s` is resized to 10, the operation will return the position of `:` in the string meaning that it will be truncated from that point.

A new tool to help us write performant string handling code.

20.5 Require span & basic_string_view to be TriviallyCopyable

P2251R1 updates the requirements the standard has for `std::span` and `std::string_view`. Starting from C++23 they must satisfy the `TriviallyCopyable` concepts.

²<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1072r10.html>

³里缪注：此处写错了，应该是`count-size()`。

As both of these objects already have default copy assignment operators and constructs and also destructors and besides they only expose a `size_t` and a raw pointer, it is implied that these types can be trivially copyable and in fact, the major compilers already implemented them as so.

Ensuring this trait for the future makes sure developers can continue depending on these characteristics and less courageous developers can start using them as such for example in heterogeneous computing.

20.6 string-stream with std::span-based buffer

C++23 is introducing the `<spanstream>` header. Streams are an old part of the C++ standard library. Nowadays **stringstreams** are widely used. Strings (and vectors) store data outside of themselves. When the data to be stored grows, the storage and the already stored data might be automatically and dynamically reallocated. This is often acceptable, but when it's not, we need another option.

`<spanstream>` is going to provide such an option, they provide fixed buffers. You have to take care of the allocation when you create your stream, but you don't have to worry about the costly reallocation of the underlying buffer once it's exhausted. When I wrote that you have to take care of the bugger allocation, I really meant it. The buffer is not owned by the stream object, its life has to be managed by the programmer.

20.7 Conclusion

I hope you enjoyed this article and you also got excited by all these various `string/string_view` related features that C++23 is going to bring us. What are you waiting for the most?

Chapter 21

C++23: Preprocessing directives

👤 Sandor Dargo 📅 2022-09-07 💬 ★

The ISO Committee accepted two proposals for C++23 related to preprocessing directives. P2334R1 is about introducing `#elifdef` and `#elififndef` and P2437R1 is introducing `#warning`.

21.1 What is a preprocessing directive?

First, let's discuss what are preprocessing or preprocessor directives. They can be included both in header and implementation files and they start with a `#` sign. Such lines are examined and resolved before the compilation starts and as the name suggests that is done by the preprocessor. Such directives are only one line long and end with a newline, so no semicolon is required. In case you want longer directives, you have to end the line with `\` in order to signal to the preprocessor that you'd continue the definition on the next line.

If you think about simple one-liner preprocessing directives, you can think about the `#include` statements. What happens for `#include` statements is that they are basically textually replaced by files being included.

Another (in)famous usage of preprocessor directives is macros! With the directive `#define` we can introduce shortcuts for literals, functions, classes, whatever. The problem is that again, it's just a textual replacement before the compilation starts. Then if you face with a syntax error or if you have to debug, you have quite a difficult job because what was compiled was textually different from what you have in your source code. But enough on why macros are bad.

If you are looking for an example for a multiline directive, it's usually a macro defined with `#define` such as

```
1 #define square(x) \
2     ((x)           \
3      *            \
4      (x))         \
```

21.2 #elifdef and #elifndef

C++23 is introducing `#elifdef` and `#elifndef`. Interestingly this is coming from WG14, so the ISO work group working on the C language. These new directives are going to be part of C23, and the C++ working group (WG21) decided to adopt these changes in order to avoid preprocessor incompatibilities with C.

In any case, it's going to help save some keystrokes, and it reads relatively well.

Probably you've seen `#ifndef` at least in some header guards. `#ifndef` identifier serving a shortcut for `#if !defined(identifier)` and there is also `#ifdef identifier` is a shorthand for `#if defined(identifier)`. In the C++ language we don't only have `if`, but there is also `else if` for supporting multibranch conditionals. It's similar in the preprocessor directives world, where we don't only have `#if`, but also `#elif`. Until now, they had no similar shorthands, but C23 and C++23 makes life easier for those who use these multibranch conditional preprocessor directives. And in fact, not just easier but also more symmetric and readable.

There is very little reason to have shorthands for one and not the other.

21.3 #warning

As I explained in *C++23: Narrowing contextual conversions to bool*, a new standard doesn't always change the compiler implementations. Sometimes, a new standard just gets closer to existing implementations.

It's the case with the introduction of `#warning` to the standard. It had been already implemented by all the major compilers, and now both the C and the C++ standard is going to adopt it.

So what does `#warning`? It invokes diagnostic message by the preprocessor, without stopping the translation. That's the difference between `#error` and `#warning`. The former stops the translation, while the latter does not.

Its usage could look like this:

```

1 #ifndef FOO
2 #warning "FOO defined, performance might be limited"
3 #endif

```

21.4 Conclusion

It might be a bit surprising, but C++23 (and C23) introduces changes to the preprocessor directives. With the introduction of `#elifdef` and `#elifndef` the language becomes more round as we already had `#ifdef` and `#ifndef` and with the standardization of `#warning`, the standard comes closer to the compiler implementations.

How often do you use preprocessor directives apart from `#include` and header guards?

Chapter 22

C++23: The stacktrace library

👤 Sandor Dargo 📅 2022-09-21 🏷️ ★

So far, there was no way in C++ to get runtime information on the current call sequence. Other popular programming languages such as Java, C# or Python provide this possibility. Thanks to P0881R7¹ and the people behind, now we will also get a similar feature with C++23.

Let's discover in this article what exactly we get and how we can use it!

22.1 Is it already available? - the meta how

Before we delve into how to use this new library, we should discuss how to use it or if we can already use it at all. I mean it's a C++23 feature and it doesn't have wide compiler support for the time being. Following a recent C++ Weekly episode², we can use the `<stacktrace>` library by compiling against at least gcc 12.1 (no trunk is needed), we have to specify `-std=c++23` and we have to add the command line option of `-lstdc++_libbacktrace` to link the library.

As such, we can have early access to this interesting new library!

22.2 What are the key features of the stacktrace library?

Let's pick some interesting and/or important decisions, features from the accepted paper:

- All `stack_frame` functions and constructors are lazy, no information will be decoded until it's needed to keep the library fast.
- Frames are stored in a dynamically sized storage as the most important piece of information is often at the bottom of the stacktrace. This also means that the stacktrace should not be constructed on performance-critical hot paths. Or at least, it should be constructed with a custom allocator.
- The `<stacktrace>` header provides us essentially with two classes. `stacktrace_entry` is the representation of one evaluation, one frame in a stacktrace and that evaluation might be empty. You can check its emptiness with `operator bool`.

¹<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0881r7.html>

²<https://www.youtube.com/watch?v=9IcxniCxKIQ>

What's in an evaluation? That's a good question. Basically, it's one entry in a backtrace or in other words a stacktrace. If you have a function `foo()` that is called from `main()`, your stacktrace should be composed of two items, two evaluations `foo()` and `main` - life can be a bit more complex though.

In order to get more information about the evaluation, you get 3 queries.

- `description()`
- `source_file()`
- `source_line()`

These names are quite self-evident, once we understand what does an evaluation of a stacktrace mean.

The other class in the header is `basic_stacktrace` and it consists of multiple stacktrace entries. It's either the representation of the full stacktrace or just a given part of it. `std::basic_stacktrace` works pretty much as a standard container with iterators and element access functions.

Keep in mind, that `stacktrace` is just an alias for `basic_stacktrace` with the default allocator.

Beware that this below piece of code might not do what you'd expect:

```

1 auto currentStacktrace = std::stacktrace(); // Won't work as one might expect!
2 for (const auto& entry : currentStacktrace) {
3     std::cout << entry.description() << std::endl;
4     std::cout << entry.source_file() << std::endl;
5     std::cout << entry.source_line() << std::endl;
6 }
```

`std::stacktrace()` is just a constructor call and it instantiates a new `basic_stacktrace` container. If you want to get the stacktrace of the current execution context, call the `current static` member function instead of the constructor.

```
auto currentStacktrace = std::stacktrace::current();
```

Once we have a stacktrace, we can obtain the stored information in different ways. The easiest way is to actually just print the whole stacktrace all at once.

```

1 #include <stacktrace>
2 #include <iostream>
3
4 void foo() {
5     auto trace = std::stacktrace::current();
6     std::cout << std::to_string(trace) << '\n';
7 }
8
9 int main() {
10     foo();
11 }
12 /*
13     0# foo() at /app/example.cpp:5
14     1#         at /app/example.cpp:10

```

```

15     2#      at :0
16     3#      at :0
17     4#
18 */

```

We see two interesting things above. `main` is not printed as a description and there are two additional frames on the top that must be related to the execution context.

As `stacktrace` is a container, we can iterate over it and print the items one by one.

```

1 #include <stacktrace>
2 #include <iostream>
3
4 void foo() {
5     auto trace = std::stacktrace::current();
6     for (const auto& entry: trace) {
7         std::cout << std::to_string(entry) << '\n';
8     }
9 }
10
11 int main() {
12     foo();
13 }
14
15 /*
16 foo() at /app/example.cpp:5
17         at /app/example.cpp:12
18         at :0
19         at :0
20
21 */

```

We have pretty much the same output, but now we lost the numbering, which would have to be put back with the help of a loop index.

If we don't want all the information from a trace, we can get the method name (description), the source file and the line number separately with the right accessors.

We can iterate over it and print each entry. We can take the different attributes:

```

1 #include <stacktrace>
2 #include <iostream>
3
4 void foo() {
5     auto trace = std::stacktrace::current();
6     for (const auto& entry: trace) {
7         std::cout << "Description: " << entry.description() << std::endl;
8         std::cout << "file: " << entry.source_file() << std::endl;

```

```

9      std::cout << "line: " << entry.source_line() << std::endl;
10     std::cout << "-----" << std::endl;
11 }
12 }
13
14 int main() {
15     foo();
16 }
17 /*
18 Description: foo()
19 file: /app/example.cpp
20 line: 5
21 -----
22 Description:
23 file: /app/example.cpp
24 line: 15
25 -----
26 Description:
27 file:
28 line: 0
29 -----
30 Description:
31 file:
32 line: 0
33 -----
34 Description:
35 file:
36 line: 0
37 -----
38 */

```

An interesting thing I found is that if the current trace is queried when a parameter is defaulted, that function doesn't appear in the stacktrace. Somehow it makes sense because it's not yet executed yet, but it was already called, so I'm not sure if I like this behaviour. But it might be just me.

```

1 #include <stacktrace>
2 #include <iostream>
3
4 void foo(std::stacktrace trace = std::stacktrace::current()) {
5     std::cout << std::to_string(trace) << '\n';
6 }
7
8 int main() {
9     foo();

```

```
10  }
11
12  /*
13      0#      at /app/example.cpp:9
14      1#      at :0
15      2#      at :0
16      3#
17  */
```

22.3 Conclusion

`<stacktrace>` library is a very useful addition to the C++ standard library that lets us query and print the backtrace. The compiler support is very limited for the time being, we can only use gcc and probably the implementation will still change here and there. Still, we can already experiment, we can already learn how to use it. I’m sure it will come in very handy for error handling in C++.

Chapter 23

C++23: flat_map, flat_set, et al.

👤 Sandor Dargo 📅 2022-10-05 🏷 ★★

C++23 is introducing some more data structures¹, some more associative containers. We are going to get the flat versions of `map`/`set`/`multimap`/`multiset`:

- `flat_map`
- `flat_set`
- `flat_multimap`
- `flat_multiset`

These new types will work as drop-in replacements for their non-flat types. The goal of these new types is to provide different time and space complexities compared to the original containers. The non-flat versions' implementations are using balanced binary trees under the hood which more-or-less defines their main characteristics. C++11 introduced the unordered versions of these containers, and even though in most cases they should be preferred, they are often neglected. As the name suggests, unordered containers are not sorted.

Now we are getting sorted containers that are more effective in a big chunk of our use cases.

As mentioned, the original containers use binary search trees, the unordered versions use hashmaps. The flat ones use sequence containers. In fact, the flat versions are not even containers, they are container adapters.

Container adapters² are interfaces created by limiting functionality in a pre-existing container and providing a different set of functionality. When you declare a container adapter, you have the option of specifying which sequence container should be the underlying container.

The underlying data structure is configurable through template parameters, but they must be sequence containers with random access iterators. Why do I speak about template parameters in plural? Because for a `flat_map`, you can use different containers for keys and values. From now on, I'll simply write about `flat_map`, but the observations are also valid for the other flat container adapters unless I explicitly write so.

¹<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0429r9.pdf>

²<https://stackoverflow.com/questions/3873802/what-are-containers-adapters-c>

23.1 Specialties of a flat_{map|set}

So what are the specialities of these container adapters over the usual associate containers? What are those different time and space complexities that it has to provide? Let's start with enumerating the disadvantages, just to avoid thinking that a `flat_map` is too good to be true.

- Insertion and deletion are going to be slower
- On insertion and deletion iterators will become unstable
- The exception safety is weaker because moves and copies do happen, we don't just pass around pointers anymore
- And we cannot store non-copyable, non-movable types in *flat* structures

And now let's see what we get for this price. We'll get:

- Faster iteration
- Random access iterators instead of bidirectional iterators
- Smaller memory consumption
- Improved cache-friendliness due to contiguous memory layout

Actually, all this comes from the fact that under a `map` you'll find a balanced search tree, while under a `flat_map`, you'll find a sequence container, like a `std::vector`.

Let's have a closer look at some of these items above.

What happens when we insert into a `map` or when we delete from it and the tree has to be rebalanced? There will be no copies or moves, because each node in the `map` has a handle, a pointer to the data and only these handles will be moved around not the pointed objects.

Once we understand this, we also take it evident that the flat versions have a better memory footprint. When you work on a contiguous memory area when you deal with a sequence container, you have no handles, no metadata to store. You simply have to deal with the data. You also have to work more on them when you insert/delete as it's not enough to move around the handles anymore.

If you want to go deeper into the cache friendliness topic, I'd recommend watching the talk of Bjorn Fahller at C++ OnSea³. He explained that even in use cases when we might think that a linked list would serve us better than a sequence container, the latter might be a better choice. Even if it has to perform more work. More work is sometimes faster as the bottleneck is not the CPU anymore, but the cache. With sequence containers, the CPU has to access contiguous parts of memory, which is particularly cache-friendly.

23.2 One more word on speed

As it was already mentioned, `flat_map` is an ordered map. If you give some inputs to it either at construction time or later, it will take care of keeping itself sorted. That requires some resources. But what if your data is already sorted? After all, C++ has the concept of not paying for what you don't use.

`flat_map` provides a solution for that!

³<https://www.youtube.com/watch?v=yyNWKHoDtMs>

The standard library provides a tag type called `sorted_unique_t` and the `flat_map` constructors have overloads taking that tag as a first parameter. Not only the constructors but also those overloads of the `insert` method that take multiple elements to insert. If you construct a `flat_map` from elements that are already sorted, or when you `insert` a range of items that are already sorted, don't forget to use the overloads with `sorted_unique_t`, because you can benefit from a significant performance gain.

But beware! If you use a `sorted_unique_t` overload with unsorted data, the behaviour is undefined! All bets are off!

23.3 How it differs in its API

`flat_map` stores separately the keys and values, and the storage for those can be of different types. Because of that, there is a bunch of new constructors available. In addition, there are several new overloads for the constructor and the `insert` method taking the previously explained `sorted_unique_t` tag so that we don't resort to already sorted items.

The `extract` member method moves out both underlying storage containers. The `extract` function is overloaded with `&&` showing that the original object should be used anymore.

The other direction of moving data is also possible through the `replace` function. It takes containers as rvalue references and replaces the underlying containers with what was passed in.

23.4 What if you want to use the new adaptors?

You'll still have to wait for the standard versions. At the moment of writing this article, no compiler supports the flat container adapters⁴.

At the same time, the proposal didn't just come out of the blue. `boost` has had this feature for quite a while⁵, which served as a basis for the standardization. You can go and experiment with it, if not on your local, you go to Compiler Explorer⁶ or Coliru⁷.

The screenshot shows the Compiler Explorer interface. On the left, a code editor displays the following C++ code:

```

1 #include <iostream>
2 #include <boost/container/flat_map.hpp>
3
4 int main () {
5     boost::container::flat_map<int, std::string> mymap {
6         {1, "one"}, {2, "two"}, {3, "three"}
7     };
8
9     for (auto [k, v] : mymap) {
10        std::cout << k << ":" << v << '\n';
11    }
12 }

```

On the right, the assembly output window shows the assembly code generated by clang 14.0.0:

```

x86-64 clang 14.0.0 (C++, Editor #1, Compiler #1) -O3 -std=c++20
main:
    push    rbp
    push    r15
    push    r14
    push    r13
    push    r12
    push    rbx
    sub     rsp, 168
    mov     dword ptr [rsp + 48], 1
    lea     r13, [rsp + 72]
    mov     dword ptr [r13 + 561], r13

```

The bottom window shows the execution results:

```

Program returned: 0
1: one
2: two
3: three

```

⁴ 里缪注：当前（2023-01-04）依旧没有编译器支持

⁵ https://www.boost.org/doc/libs/1_80_0/doc/html/boost_container_header_reference.html#header.boost.container.flat_map_hpp

⁶ <https://godbolt.org/z/x3vY1f6rz>

⁷ <http://coliru.stacked-crooked.com/a/84be54a775297036>

23.5 Conclusion

In C++23, we are going to get some exciting new container adaptors in the standard library! The `flat_{map|multimap|set|multiset}` containers offer different space and time complexities compared to their normal, original versions. It favors fast iteration, lookup and lower memory consumption at the expense of potentially slower writes. At the same time, it still offers the benefits of having sorted containers, unlike the `unordered_*` versions.

Although there is no compiler support for the time being, we can already learn about both from the standard or by using the boost versions!

Chapter 24

C++23: How lambdas are going to change?

• Sandor Dargo  2022-11-23 

C++23 is coming soon and it will change how lambdas work in 3 different ways. They will not only become simpler in certain circumstances but they will be also aligned more with other features of the language.

Okay, let's go through these changes.

24.1 Make () more optional for lambdas

What is the simplest lambda function?

Is it `[]()`?

Nope! It's `[]`.

If a lambda has no parameters (in standardese: if the parameter declaration clause is empty), we can omit the parentheses!

Or can't we?

It depends. In regular lambda functions we can, but there has been another not-very-intuitive rule. When you have some lambda template parameters, `constexpr`, `consteval`, `mutable` or `noexcept` specifiers, or when you benefit from attributes or trailing return types (see the next section!) or a `requires` clause, you cannot omit the empty parentheses.

So the following lambda declaration is illegal in C++20:

```
// warning: parameter declaration before lambda declaration specifiers only optional
//   with '-std=c++2b' or '-std=gnu++2b' [-Wc++23-extensions]
auto l = [] mutable {};
```

Thanks to P1102R2¹, omitting empty parentheses in such circumstances becomes legitim. A small proposal that doesn't change how existing code behaves but makes the language a bit more comfortable and more importantly, it makes the language more consistent.

¹<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1102r2.html>

24.2 Change the scope of lambda trailing-return-type

I found overly interesting this change and it definitely made me learn something important I had no idea about!

Let's assume that you have such a piece of code:

```

1 double j = 42.0;
2 // ...
3 auto counter = [j=0]() mutable -> decltype(j) {
4     return j++;
5 };

```

I know the trailing return type makes not much sense here, but this is just a simple example to showcase the problem.

What do you expect the return type of `counter` to be?

You might argue that in the lambda capture we declare `j` and initialize it with `0`. The `0` literal without any suffix is an int, therefore when we invoke `counter` it should return an `int`.

That's wrong, the right answer is `double`!

Even though the `j` introduced in the capture is physically closer to the trailing return type and most probably developers would first think about the captured `j` rather than some other variable from the outer scope, still the answer is that the type of `j` comes from the outer scope. **Whatever is in the capture list, it's not visible in the trailing return type.**

This also means that the following snippet on its own would not compile:

```

1 auto counter = [j=0]() mutable -> decltype(j) {
2     return j++;
3 };

```

To be fair, this is the luckier case. It's better to have a clear error rather than an unexpected and often undetected behaviour like in the original example.

P2036R3² is going to change this situation. Name lookups for trailing return types are going to consider captures before looking outside. While this is not going to be a backward-compatible change, it will almost always match the developer's intent.

24.3 Attributes on lambdas

After reading this section title, you might ask whether it's already possible to use attributes with lambdas. The answer is yes, it is possible. The place for adding an attribute is in the lambda declarator, either before or after the parameter declaration clause, but always between the optional noexcept specifiers and trailing return types.

The attributes sequence belongs to the type of the corresponding function call operator. The paper P2173R1³ argues - and rightly so - that this should not necessarily be the case. Attributes should be allowed to belong to the function call operator.

²<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2036r3.html>

³<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2173r1.pdf>

After all, why couldn't the operator and with that almost always the lambda be `[[nodiscard]]`, `[[noreturn]]` or `[[deprecated]]`?

According to the proposed and accepted change, regardless of the other optional elements of a lambda expression, now we can declare the attribute sequence right after the lambda introducer (and its optional capture) or right after the template parameter list including its requires clause.

The proposed wording says that an attribute specifier sequence in a lambda declarator appertains to the type of the function call operator, but if it comes before the lambda declarator, then it belongs to the function call operator itself, not its type. You might say that you cannot attach a `[[nodiscard]]` attribute to a type, it wouldn't make sense. You're right, but don't think only about the standard attributes, even the proposal mentions vendor-defined ones.

With this change, the following piece of code becomes valid, where the function call of the operator is `[[nodiscard]]`:

```
auto lm = [] [[nodiscard]] ()->int { return 42; };
```

Meaning that this would emit a warning:

```
auto lm = [] [[nodiscard]] ()->int { return 42; };
// ...
// warning: ignoring return value of 'main()::<lambda()>',
//   declared with attribute 'nodiscard' [-Wunused-result]
lm();
```

Please note that GCC and Clang already implemented this behaviour, GCC already in version 9!

24.4 Conclusion

Lambdas were introduced in C++11 and each standard brought some new features. It's not going to be different with C++23. It will bring better attributes, a more reasonable trailing return type deduction and more consistent rules for omitting an empty parameter list. Stay tuned for more articles about the coming standard!

Chapter 25

C++23: attributes

👤 Sandor Dargo 📅 2022-12-14 🏷️ ★

C++11 introduced attributes. Even though the language itself only came with two (`[[noreturn]]` and `[[carries_dependency]]`), it also provided the unified standard syntax for *implementation-defined language extensions*. Ever since that version, each new release brought us some new standard attributes or changes in the ways we can use them.

The list of changes continues with C++23. Three papers affect attributes. Duplications will not be punished anymore, their usage with lambdas evolves and there is also a new standard attribute `[[assume]]`. Let's have a deeper look.

25.1 DR: Allow Duplicate Attributes

So far the standard didn't allow that an attribute such as `[[nodiscard]]` appear more than once in an attribute list.

Wait? Isn't that a good thing? You might ask these questions, and that's exactly what I asked myself when I saw this DR proposal.

When you see DR in a proposal title, it means that it's a remediation for a "defect report" that was filed against the standard.

Defining the same attribute more than once is not a good practice. It's something you shouldn't do as a developer. But according to the paper P2156R1¹, it can happen that a macro-based solution generates the same attribute more than once.

Specifying that attributes shouldn't appear more than once is a constraint for such a solution. At the same time, it doesn't make standardization easier. On the contrary, removing this constraint, makes the standard easier! So far, it was specified for each standard attribute that "*it shall appear at most once in each attribute list*". From now on, they can and that constraint is removed from the text.

¹<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2156r1.pdf>

25.2 Attributes on lambdas

After reading this section title, you might ask whether it's already possible to use attributes with lambdas. The answer is yes, it is possible. The place for adding an attribute is in the lambda declarator, either before or after the parameter declaration clause, but always between the optional noexcept specifiers and trailing return types.

The attributes sequence belongs to the **type** of the corresponding function call operator. The paper P2173R1² argues - and rightly so - that this should not necessarily be the case. Attributes should be allowed to belong to the function call operator.

After all, why couldn't the operator and with that almost always the lambda be **[[nodiscard]]**, **[[noreturn]]** or **[[deprecated]]**?

According to the proposed and accepted change, regardless of the other optional elements of a lambda expression, now we can declare the attribute sequence right after the lambda introducer (and its optional capture) or right after the template parameter list including its requires clause.

The proposed wording says that an attribute specifier sequence in a lambda declarator appertains to the type of the function call operator, but if it comes before the lambda declarator, then it belongs to the function call operator itself, not its type. You might say that you cannot attach a **[[nodiscard]]** attribute to a type, it wouldn't make sense. You're right, but don't think only about the standard attributes, even the proposal mentions vendor-defined ones.

With this change(P2173R1), the following piece of code becomes valid, where the function call of the operator is **[[nodiscard]]**:

```
auto lm = [] [[nodiscard]] ()->int { return 42; };
```

Meaning that this would emit a warning:

```
1 auto lm = [] [[nodiscard]] ()->int { return 42; };
2 // ...
3 // warning: ignoring return value of 'main()::<lambda()>', declared
4 //           with attribute 'nodiscard' [-Wunused-result]
5 lm();
```

Please note that GCC and Clang already implemented this behaviour, GCC already in version 9!

25.3 Attribute **[[assume]]**

The new attribute **[[assume]]** is the standardized version of some already existing compiler-specific attributes, such as **__assume()** (MSVC, ICC) and **__builtin_assume()** (Clang). With their help, the programmer can signal to the compiler that an expression will be true without the compiler having to evaluate it.

On gcc, an alternative was not available, so probably that's why they are the first, who implemented the standardized **[[assume]]** in version 13. If you want to emulate the same functionality before you have to use **if(expr) {} else {__builtin_unreachable();}**.

²<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2173r1.pdf>

As such, the compiler can generate more efficient, faster code. This can bring significant benefits in some high-performance, low-latency applications.

It's syntax is `[[assume(expr)]]`. It's worth noting that the expression is not evaluated. Which is a change compared to the gcc simulation, but is the same as how clang and msvc already worked.

The accepted paper³ brings us quite a few examples when and how to use `[[assume]]`. The first one is an example from audio processing, and it uses information that is commonly known by the developers, but it's not represented in the code:

```

1 void limiter(float* data, size_t size) {
2     [[assume(size > 0)]]; 
3     [[assume(size % 32 == 0)]]; 
4
5     for (size_t i = 0; i < size; ++i) {
6         [[assume(std::isfinite(data[i]))]];
7         data[i] = std::clamp(data[i], -1.0f, 1.0f);
8     }
9 }
```

I first overlooked the first assumption, but Thief in the comments and Julien in an e-mail gently pointed out my mistake. The first assumption is that the parameter `size` will never be zero, but it will always be a positive number!

The second example hints to the compiler that the passed in size will always be the multiple of 32 and with the last assumption, we even tell that the data is not `NaN` or `infinity`.

In another example, the copy constructor of a reference counting shared pointer got the assumption, that the refcount is already at least one. Makes sense, it's a copy constructor. This could also help the compiler to ignore some increments and decrements and avoid destroying the owned resource when the passed in smart pointer's lifecycle ends. Interestingly, it didn't lead to great optimizations on some of the compilers.

So I'd say, measure, measure and measure before you assume some gains.

On most compilers, this leads to a significant performance increase (for the details, please check paper P1774R8).

But what if an assumption does not hold true?

It's important to emphasize that assumptions are not evaluated! They are not checked! Instead, the compiler assumes that the expression **would** evaluate to true and optimize accordingly. If the assumption does not hold, so if the assumption returns false, throws an exception or if it's UB, the behaviour will be undefined. Use it with care.

25.4 Conclusion

C++23 brings 3 changes to attributes. The restriction that an attribute cannot be duplicated is removed now. Attributes can be used with lambdas in a way that it belongs to the function call operator from now on, not to the lambda itself. Last but least, we get a new standard attribute `[[assume]]` to ease compiler optimizations.

Do you often use attributes?

³<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1774r8.pdf>

Chapter 26

A Gentle Introduction to `mdspan`

👤 Mark Hoemmen 📅 2022-11-11 🏷️ ★★★

ISO-C++ standard proposals are not the best tutorials for how to use their proposed features. This is intentional—there's enough content to be written in exploring the technical design issues that a tutorial-style introduction is out of place in such documents. However, it also means that proposals such as P0009 for `mdspan` reach a level of maturity where they are ready for production implementation and use, but without a single tutorial-style introduction. Here's an attempt to do so for `mdspan`.

26.1 What is `mdspan`?

In its simplest form, `std::mdspan` is a straightforward extension of `std::span` to enable multidimensional arrays. (The `std::span` class was added in C++20, and introductions to it can be found all over the internet. Thus, this tutorial assumes basic familiarity with `std::span`.) You can create them from a pointer to the underlying data, and its extents (the dimensions of the multidimensional array).

```
1 int* data = /* ... */  
2  
3 // View data as contiguous memory representing 4 ints  
4 auto s = std::span(data, 4);  
5  
6 // View data as contiguous memory representing 2 rows  
7 // of 2 ints each  
8 auto ms = std::mdspan(data, 2, 2);
```

In the example above, we use CTAD (constructor template argument deduction, a language feature introduced in C++17) so that we don't have to name the template arguments of the `mdspan` class. The compiler figures them out for us.

Accessing the data then happens via its `[]` operator, just as with `span` or `vector`.

```
for(size_t i=0; i<ms.extent(0); i++)  
for(size_t j=0; j<ms.extent(1); j++)  
    ms[i, j] = i*1000 + j;
```

(Note that the `mdspan` implementation in this repository uses `()` if the compiler doesn't support multi-argument `[]` - a feature which was voted into C++23.) `mdspans extent` function takes an index to indicate which `extent` the caller wants.

26.2 Extents

Just like `span`, you can use both dynamic extents and static extents with `mdspan`. Unlike `span`, `mdspan` is not directly templated on individual extents, but packs all of them into an explicit `extents` object. An `extents` object represents the "shape" of a multidimensional array. It can encode any number and combination of extents as compile-time values (that are part of the type of the `extents` object), and encodes the remaining extents as run-time values (that must be given as arguments to the `extents` constructor).

```

1 // Create an extents object with one run-time dimension 3,
2 // and one static (compile-time) extent 4.
3 // It represents the shape of a 3 x 4 array.
4 std::extents<int, std::dynamic_extent, 4> exts(3);

```

The `extents` class does not only take the actual sizes (or extents) of each dimension, it also takes as an argument the `index_type`, the `extents` class - and through it `mdspan` - will use to store the extents, and do index computation. The total number of arguments after the `index_type` is also the `rank` of the `extents` object, and later the `rank` of the `mdspan`.

You can construct an `mdspan` by providing the individual extents to the constructor (either all of them, or just the dynamic ones), or handing in an `extents` object.

```

1 int* data = /* ... */
2 int size = /* ... */
3
4 int rows = /* ... */
5 constexpr int cols = 4;
6
7 // An extents type using uint32_t to represent each extent,
8 // with a run-time number of rows, and 4 columns.
9 using ext_t = std::extents<uint32_t, std::dynamic_extent, cols>;
10
11 // When creating the mdspan, we only need to give run-time extents.
12 // The compile-time extents are already baked into the extents type.
13 auto ms = std::mdspan<int, ext_t>(data, rows);
14 assert(ms.extents(0) == rows);
15 assert(ms.extents(1) == cols);
16 static_assert(ms.static_extent(1) == cols);
17 static_assert(decltype(ms)::rank() == 2);
18
19 // If we don't specify the mdspan's template parameters,
20 // then we need to pass in all the extents.

```

```

21 // It will then store all extents as run-time values.
22 auto msd = std::mdspan(data, rows, cols);
23 assert(msd.extents(0) == rows);
24 assert(msd.extents(1) == cols);
25 static_assert(decltype(msd)::rank() == 2);

```

Note that `dextents` is a convenient alias for an `extents` type with all `dynamic_extent`.

26.3 Layout and access customization

The design of `mdspan` addresses a much broader variety of needs than `span`. In many scenarios, it even makes more sense to use a one-dimensional `mdspan` instead of a `span`. This broader scope of use cases is addressed through two customization points: `LayoutPolicy` and `Accessor`.

```

1 template <
2     class T,
3     class Extents,
4     class LayoutPolicy = std::layout_right,
5     class Accessor = std::default_accessor
6 >
7 class mdspan;

```

The `LayoutPolicy` defines the data layout, that is, the function that turns a multi-dimensional index i, j, k, \dots into a one-dimensional offset. The `Accessor` defines how the array stores its elements, and how to use the offset from the `LayoutPolicy` to get a reference to an element of the array. (For example, the default `Accessor` takes a pointer `p` and an offset `i`, and returns `p[i]`.)

You can think of these customizations like the same kind of thing as the `Hash` template parameter on `std::unordered_map` or the `Allocator` template parameter on `std::vector`. These are things that most of the time you don't have to touch, and that most algorithms shouldn't have to care about. (How many times have you written a function that takes a `std::vector` and had to write a different version of the function for different allocators? Probably not that often, if ever.)

26.4 Basics of LayoutPolicy

As stated above the `LayoutPolicy` controls the mapping from a multi-dimensional index into an offset. One of the reasons why controlling a layout mapping can be useful, is to control an algorithm's data access pattern (how it strides through memory) without changing the algorithm's loop structure.

For example, it's natural to think of a matrix-vector multiply algorithm as a loop over rows, with an inner loop which performs the inner reduction:

```

1 for(int i = 0; i < A.extent(0); i++) {
2     double sum = 0.;
3     for(int j = 0; j < A.extent(1); j++)
4         sum += A[i, j] * x[j];

```

```

5     y[i] = sum;
6 }
```

In most cases an optimal data access pattern is achieved if the stride-one access is on `j`. This also allows for vectorization of the inner loop. However, consider the case where `A.extent(1)` is small and known at compile time. In that case it might actually be better to have the inner loop fully unrolled, and have a stride-one access on `i` instead, so that the outer loop can be vectorized.

```

1 for(int i = 0; i < A.extent(0); i++) {
2     y[i] = A[i, 0] * x[0] +
3             A[i, 1] * x[1] +
4             A[i, 2] * x[2] +
5             A[i, 3] * x[3];
6 }
```

With `mdspan` the first use case can be achieved with `layout_right` (which is the default) and the second use case might perform better with `layout_left`:

```

1 mspan<double, extents<uint32_t, dynamic_extent, 4>, layout_left> A(A_ptr, N);
2 for(int i = 0; i < A.extent(0); i++) {
3     y[i] = A[i, 0] * x[0] +
4             A[i, 1] * x[1] +
5             A[i, 2] * x[2] +
6             A[i, 3] * x[3];
7 }
```

Another important use case where layouts are needed is for reusing functionality for a subset of data.

For example one might want to perform an operation on a single column of some multi-dimensional array. That column, however, may not be a contiguous array. `layout_stride` can deal with that in many common cases:

```

1 mspan A(A_ptr, N, M);
2 for(int c = 0; c < A.extent(1); c++) {
3     extents sub_ext(A.extents(0));
4     array strides{A.stride(0)};
5     layout_right::mapping sub_map(sub_ext, strides);
6     mspan col_c(&A(0,c), sub_map);
7     foo_1d_mspan(col_c);
8 }
```

A more advanced tutorial will cover the topics of layout and access customization, but for now, it's enough to know that two simple ones are provided: `std::layout_right` and `std::layout_left`, where the right-most and left-most indices are fast-running, respectively. When discussing matrices, these are often described as C-style layout (or row-major) and Fortran-style (or column-major), respectively.

26.5 Views and ownership

We say that `mdspan` is a "view." What does this mean, and how does it relate to ownership?

”Ownership” refers to who frees the memory. ”Nonowning” means ”it doesn’t free the memory.”

By convention, `const T*` and `T*` are nonowning¹. `span<T>` and `span<const T>` are nonowning. However, `vector<T>` is owning. When the `vector` goes away, it frees memory. Standard containers are owning.

”View” means that if you have a view X, and you make a copy of X called Y, then X and Y refer to the same elements. Views don’t copy elements. Copying views should also be cheap; it shouldn’t depend on the number of elements being viewed. (Ranges has some subtle definition of ”view” that permits some exceptions, but this gets at the essence.)

Pointers are views. `span` is a view. `mdspan` is a view. `vector` is not a view; it’s a container. If you make a copy of the vector, it will copy all the elements. The elements in the new vector are different elements (”different” in a similar sense as meant by ”regular type”) than the old vector’s elements.

`std::shared_ptr` is a view, EVEN THOUGH it expresses shared ownership. This matters because `mdspan` is a view, BUT whether it is owning depends on the accessor. With `default_accessor`, the `mdspan` is non-owning. However, you could define an accessor whose `data_handle_type` is the following.

```

1 struct shared_ownership_data_handle {
2     shared_ptr<element_type[]> base;
3     // This mdspan's data start at base[offset].
4     // This exists because shared_ptr<T[]> has no "offset constructor"
5     // that expresses shared ownership of an array through a subset.
6     size_t offset;
7 };

```

That would make different instances of `mdspan` share ownership of the allocation. When the last ”shared owner” falls out of scope, the memory would be deallocated.

¹<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-ptr>

Chapter 27

C++23's new function syntax

👤 Ben Deane 📅 2022-09-05 🔮 ★

We've had a couple of ways to spell functions for a long time:

```
1 [[nodiscard]] auto say_a_to(
2     std::string_view what, std::string_view name) -> std::string {
3     return std::string{what} + ", " + std::string{name} + "!";
4 }
5
6 say_a_to("Hi", "Kate"); // -> "Hi, Kate!"
7
8 struct {
9     [[nodiscard]] auto operator()(

10        std::string_view what, std::string_view name) const -> std::string {
11        return std::string{what} + ", " + std::string{name} + "!";
12    }
13 } say_b_to;
14
15 say_b_to("Hello", "Tony"); // -> "Hello, Tony!"
```

And we've had a shortcut for that second one for a while, too:

```
1 auto say_c_to = [] [[nodiscard]] (
2     std::string_view what, std::string_view name) -> std::string {
3     return std::string{what} + ", " + std::string{name} + "!";
4 };
5
6 say_c_to("Bye", "Kate"); // -> "Bye, Kate!"
```

(Aside: notice where `[[nodiscard]]` is positioned, since C++23's P2173, to apply to the lambda's call operator.)

All these ways to spell functions look the same at the call site. But “regular functions” and function objects behave a bit differently, so as application and library writers we use them for different things according to our needs –how we need users to customize or overload them; whether we want ADL; etc.

C++23 added a new way to spell functions:

```

1 struct {
2     [[nodiscard]] auto operator[](
3         std::string_view what, std::string_view name) const -> std::string {
4             return std::string{what} + ", " + std::string{name} + "!";
5     }
6 } say_d_to;
7
8 say_d_to["Goodbye", "Tony"]; // -> "Goodbye, Tony!"

```

It hasn't been publicized as such, but that's exactly what it is. C++20 removed the ability to use the comma operator in subscripts (P1161). As a follow-up to that, C++23 now allows multi-argument subscript operators (P2128). And that effectively gives us an alternative syntax for calling functions. The subscript operator now has the same mechanics as the function call operator. These operators are now the only two that can take arbitrary numbers of arguments of arbitrary types. And they're also the same precedence.

So we can do things like this too:

```

1 struct {
2     template <std::integral ...Ts>
3     [[nodiscard]] auto operator[](Ts... ts) const noexcept {
4         return (0 + ... + ts);
5     }
6 } sum;

```

And call it accordingly:

```

1 const auto s1 = sum[1, 2, 3]; // 6
2 const auto s2 = sum[];      // 0

```

(Yes, `operator[]` can also be written as a nullary function.)

This works, today. Probably because to the compiler this was already the bread-and-butter of how functions work anyway, so I'm guessing (although IANACW) it was pretty easy to implement these papers.

P2128's envisaged use cases are all about numeric computing and multi-dimensional arrays with integral subscripting. But that's not all that `operator[]` is now. Quite literally, it's alternative syntax for a function call, with everything that might imply.

What use might this be? Well a few things spring to mind. Using `operator[]` for function calls has all the same lookup and customization implications as using `operator()`, but adds the inability to call through type-erased function wrappers —at least at the moment. So that might be useful to someone.

A second convention that springs to mind is perhaps for pure functions. If a function is “pure” then it will always return the same output given the same input, which means mathematically it can be implemented with a lookup table. Using `operator[]` historically looks something like a map lookup, so perhaps it's a natural fit for pure function syntax?

It might also be useful to naturally express two different areas of functionality within a library, or operations with different evaluation semantics (compile-time? runtime? lazy?) as characterized by function calls with `operator()` and with `operator[]`. This would perhaps provide a nice call-site indication to make the code more readable.

There are sure to be other uses. Should you look for `operator[]` coming soon to a library near you? I don’t know. This might seem strange to some folks, but it’s not necessarily less readable; just less familiar. And if there’s one thing I know about C++, it’s that it’s a short hop from bizarre newly-discovered quirk to established technique. `operator[]` is now equivalent to `operator()`, and when someone finds a use for that, it will get used.

Chapter 28

Functional exception-less error handling with C++23's optional and expected

• Sy Brand 📅 2023-04-18 💬 ★★

This post is an updated version of one I made over five years ago¹, now that everything I talked about is in the standard and implemented in Visual Studio.

In software things can go wrong. Sometimes we might expect them to go wrong. Sometimes it's a surprise. In most cases we want to build in some way of handling these misfortunes. Let's call them disappointments².

`std::optional` was added in C++17³ to provide a new standard way of expressing disappointments and more, and it has been extended in C++23 with a new interface inspired by functional programming⁴.

`std::optional<T>` expresses “either a `T` or nothing”. C++23 comes with a new type, `std::expected<T,E>`⁵ which expresses “either the expected `T`, or some `E` telling you what went wrong”. This type also comes with that special new functional interface⁶. As of Visual Studio 2022 version 17.6 Preview 3, all of these features are available in our standard library. Armed with an STL implementation you can try yourself, I'm going to exhibit how to use `std::optional`'s new interface, and the new `std::expected` to handle disappointments.

One way to express and handle disappointments is exceptions:

```
1 void pet_cat() {
2     try {
3         auto cat = find_cat();
4         scratch_behind_ears(cat);
5     }
6     catch (const no_cat_found& err) {
7         //oh no
```

¹<https://blog.tartanllama.xyz/optional-expected/>

²<https://wg21.link/p0157>

³<https://devblogs.microsoft.com/cppblog/stdoptional-how-when-and-why/>

⁴<https://wg21.link/p0798>

⁵<https://wg21.link/P0323>

⁶<https://wg21.link/p2505>

```

8     be_sad();
9 }
10 }
```

There are a myriad of discussions, resources, rants, tirades, and debates about the value of exceptions⁷, and I will not repeat them here. Suffice to say that there are cases in which exceptions are not the best tool for the job. For the sake of being uncontroversial, I'll take the example of disappointments which are expected within reasonable use of an API.

The Internet loves cats. Suppose that you and I are involved in the business of producing the cutest images of cats the world has ever seen. We have produced a high-quality C++ library geared towards this sole aim, and we want it to be at the bleeding edge of modern C++.

A common operation in feline cutification programs is to locate cats in a given image. How should we express this in our API? One option is exceptions:

```
// Throws no_cat_found if a cat is not found.
image_view find_cat (image_view img);
```

This function takes a view of an image and returns a smaller view which contains the first cat it finds. If it does not find a cat, then it throws an exception. If we're going to be giving this function a million images, half of which do not contain cats, then that's a lot of exceptions being thrown. In fact, we're pretty much using exceptions for control flow at that point, which is A Bad Thing™.

What we really want to express is a function which either returns a cat if it finds one, or it returns nothing. Enter `std::optional`.

```
std::optional<image_view> find_cat (image_view img);
```

`std::optional` was introduced in C++17 for representing a value which may or may not be present. It is intended to be a vocabulary type —i.e. the canonical choice for expressing some concept in your code. The difference between this signature and the previous one is powerful; we've moved the description of what happens on an error from the documentation into the type system. Now it's impossible for the user to forget to read the docs, because the compiler is reading them for us, and you can be sure that it'll shout at you if you use the type incorrectly.

Now we're ready to use our `find_cat` function along with some other friends from our library to make embarrassingly adorable pictures of cats:

```

1 std::optional<image_view> get_cute_cat (image_view img) {
2     auto cropped = find_cat(img);
3     if (!cropped) {
4         return std::nullopt;
5     }
6
7     auto with_tie = add_bow_tie(*cropped);
8     if (!with_tie) {
9         return std::nullopt;
10    }
```

⁷<https://stackoverflow.com/questions/13835817/are-exceptions-in-c-really-slow>

```

11
12     auto with_sparkles = make_eyes_sparkle(*with_tie);
13     if (!with_sparkles) {
14         return std::nullopt;
15     }
16
17     return add_rainbow(make_smaller(*with_sparkles));
18 }
```

Well this is...okay. The user is made to explicitly handle what happens in case of an error, so they can't forget about it, which is good. But there are two issues with this:

1. There's no information about why the operations failed.
2. There's too much noise; error handling dominates the logic of the code.

I'll address these two points in turn.

28.1 Why did something fail?

`std::optional` is great for expressing that some operation produced no value, but it gives us no information to help us understand why this occurred; we're left to use whatever context we have available, or (please, no) output parameters. What we want is a type which either contains a value, or contains some information about why the value isn't there. This is called `std::expected`.

With `std::expected` our code might look like this:

```

1 std::expected<image_view, error_code> get_cute_cat (image_view img) {
2     auto cropped = find_cat(img);
3     if (!cropped) {
4         return no_cat_found;
5     }
6
7     auto with_tie = add_bow_tie(*cropped);
8     if (!with_tie) {
9         return cannot_see_neck;
10    }
11
12    auto with_sparkles = make_eyes_sparkle(*with_tie);
13    if (!with_sparkles) {
14        return cat_has_eyes_shut;
15    }
16
17    return add_rainbow(make_smaller(*with_sparkles));
18 }
```

Now when we call `get_cute_cat` and don't get a lovely image back, we have some useful information to report to the user as to why we got into this situation.

28.2 Noisy error handling

Unfortunately, with both the `std::optional` and `std::expected` versions, there's still a lot of noise. This is a disappointing solution to handling disappointments.

What we really want is a way to express the operations we want to carry out while pushing the disappointment handling off to the side. As is becoming increasingly trendy in the world of C++, we'll look to the world of functional programming for help. In this case, the help comes in the form of `transform` and `and_then`.

If we have some `std::optional` and we want to carry out some operation on it if and only if there's a value stored, then we can use `transform`:

```

1 cat make_cuter(cat);
2
3 std::optional<cat> result = maybe_get_cat().transform(make_cuter);
4 //use result

```

This code is roughly equivalent to:

```

1 cat make_cuter(cat);
2
3 auto opt_cat = maybe_get_cat();
4 if (opt_cat) {
5     cat result = make_cuter(*opt_cat);
6     //use result
7 }

```

If we want to carry out some operation which could itself fail then we can use `and_then`:

```

1 std::optional<cat> maybe_make_cuter (cat);
2
3 std::optional<cat> result = maybe_get_cat().and_then(maybe_make_cuter);
4 //use result

```

This code is roughly equivalent to:

```

1 std::optional<cat> maybe_make_cuter (const cat&);
2
3 auto opt_cat = maybe_get_cat();
4 if (opt_cat) {
5     std::optional<cat> result = maybe_make_cuter(*opt_cat);
6     //use result
7 }

```

`and_then` and `transform` for `expected` act in much the same way as for `optional`: if there is an expected value then the given function will be called with that value, otherwise the stored unexpected value will be returned. Additionally, there is a `transform_error` function which allows mapping functions over unexpected values.

The real power of these functions comes when we begin to chain operations together. Let's look at that original `get_cute_cat` implementation again:

```

1 std::optional<image_view> get_cute_cat (image_view img) {
2     auto cropped = find_cat(img);
3     if (!cropped) {
4         return std::nullopt;
5     }
6
7     auto with_tie = add_bow_tie(*cropped);
8     if (!with_tie) {
9         return std::nullopt;
10    }
11
12    auto with_sparkles = make_eyes_sparkle(*with_tie);
13    if (!with_sparkles) {
14        return std::nullopt;
15    }
16
17    return add_rainbow(make_smaller(*with_sparkles));
18 }
```

With `transform` and `and_then`, our code transforms into this:

```

1 std::optional<image_view> get_cute_cat (image_view img) {
2     return find_cat(img)
3             .and_then(add_bow_tie)
4             .and_then(make_eyes_sparkle)
5             .transform(make_smaller)
6             .transform(add_rainbow);
7 }
```

With these two functions we've successfully pushed the error handling off to the side, allowing us to express a series of operations which may fail without interrupting the flow of logic to test an `optional`. For more discussion about this code and the equivalent exception-based code, I'd recommend reading Vittorio Romeo's "Why choose sum types over exceptions?"⁸ article.

28.3 A theoretical aside

I didn't make up `transform` and `and_then` off the top of my head; other languages have had equivalent features for a long time, and the theoretical concepts are common subjects in Category Theory⁹.

I won't attempt to explain all the relevant concepts in this post, as others have done it far better than I could. The basic idea is that `transform` comes from the concept of a functor, and `and_then` comes from

⁸https://vittorioromeo.info/index/blog/adts_over_exceptions.html

⁹<https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/>

monads. These two functions are called `fmap` and `>>=` (bind) in Haskell. The best description of these concepts which I have read is *Functors, Applicatives, And Monads In Pictures* by Aditya Bhargava. Give it a read if you'd like to learn more about these ideas.

28.4 A note on overload sets

One use-case which is annoyingly verbose is passing overloaded functions to `transform` or `and_then`. For example:

```
1 cat make_cuter(cat);
2
3 std::optional<cat> c;
4 auto cute_cat = c.transform(make_cuter);
```

The above code works fine. But as soon as we add another overload to `make_cuter`:

```
1 cat make_cuter(cat);
2 dog make_cuter(dog);
3
4 std::optional<cat> c;
5 auto cute_cat = c.transform(make_cuter);
```

then it fails to compile, because it's not clear which overload we want to pass to transform.

One solution for this is to use a generic lambda:

```
1 std::optional<cat> c;
2 auto cute_cat = c.transform([](auto x) { return make_cuter(x); });
```

Another is a `LIFT` macro:

```
1 #define FWD(...) std::forward<decltype(__VA_ARGS__)>(__VA_ARGS__)
2 #define LIFT(f) \
3     [](auto&&... xs) noexcept(noexcept(f(FWD(xs)...))) -> decltype(f(FWD(xs)...)) \
4     { return f(FWD(xs)...); }
5
6 std::optional<cat> c;
7 auto cute_cat = c.transform(LIFT(make_cuter));
```

Personally I hope to see overload set lifting¹⁰ in some form get into the standard so that we don't need to bother with the above solutions.

If you want to read more about specifically this problem, I have a whole blog post¹¹ on it.

28.5 Try them out

The functional extensions to `std::expected` and `std::optional` are available in Visual Studio 2022 version 17.6 Preview 3. Please try them out and let us know what you think! If you have any questions, com-

¹⁰<https://wg21.link/p0834>

¹¹<https://blog.tartanllama.xyz/passing-overload-sets/>

ments, or issues with the features, you can comment below, or reach us via email at visualcpp@microsoft.com or via Twitter at @VisualC.

If you’re stuck on old versions of C++, I have written implementations of `optional` and `expected` with the functional interfaces as single-header libraries, released under the CC0¹² license. You can find them at `tl::optional`¹³ and `tl::expected`¹⁴.

¹²<https://creativecommons.org/share-your-work/public-domain/cc0/>

¹³<https://github.com/TartanLlama/optional>

¹⁴<https://github.com/TartanLlama/expected>

Chapter 29

The evolution of enums

👤 Sandor Dargo 📅 2023-02-15 💬 ★★

Constants are great. Types are great. Constants of a specific type are really great. This is why `enum classes` are just fantastic.

Last year, we talked about why we should avoid using boolean function parameters¹. One of the solutions proposed uses strong types, in particular using `enums` instead of raw booleans. This time, let's see how `enums` and the related support evolved during the course of the years.

29.1 Unscoped enumerations

Enumerations are part of the original C++ language, in fact, they were taken over from C. Enumerations are distinct types with a restricted range of values. The range of values is restricted to some explicitly named constants. Let's quickly have a look at an `enum`.

```
enum Color { red, green, blue };
```

After having read this very small example, it's worth noticing two things:

- The `enum` itself is a singular noun, even though it enumerates multiple values. We use such conventions because we keep in mind that it will be always used with one value. If you take a `Color` function parameter, one colour will be taken. When you compare against a value, you'll compare against one value. E.g. it reads better to compare against `Color::red` than against `Colors::red`
- The enumerator values are not written in ALL_CAPS! Even though there is a fair chance that you are used to that. I also used to do that. So why didn't I follow that practice? Because for writing this article, I checked the core guidelines and `Enum.5`² clearly says that we should not use ALL_CAPS in order to avoid clashes with macros. By the way, `Enum.1`³ clearly said that we should use enumerations over macros.

Since C++11, the number of possibilities to declare an `enum` grew. C++11 introduced the possibility of specifying the underlying type of an enum. If it's left undefined, the underlying type is implementation-defined but in any case, it's an integral type.

¹<https://www.sandordargo.com/blog/2022/04/06/use-strong-types-instead-booleans>

²https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#enum5-dont-use-all_caps-for-enumerators

³https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#enum5-dont-use-all_caps-for-enumerators

How to specify it? Syntax-wise it might seem a bit like inheritance! Though there are no access levels to define.

```
enum Color : int { red, green, blue };
```

With that you can be sure what the underlying type is. Still, it must be an integral type! For example, it cannot be a string. Should you try that and you'll get a very explicit error message:

```
main.cpp:4:19: error: underlying type 'std::string' {aka 'std::__cxx11::basic_string<char>'}
        of 'Color' must be an integral type
4 | enum Color : std::string { red, green, blue };
   |           ^~~~~~
```

Note that the core guidelines advocate against this practice⁴! You should only specify the underlying value if it is necessary.

Why can it be necessary? It gives us two reasons.

- If you know that the number of choices will be very limited and you want to save a bit of memory:

```
enum Direction : char { north, south, east, west,
    northeast, northwest, southeast, southwest };
```

- Or if you happen to forward declare an `enum` then you also must declare the type:

```
enum Direction : char;
void navigate(Direction d);

enum Direction : char { north, south, east, west,
    northeast, northwest, southeast, southwest };
```

You can also specify the exact value of one or all the enumerated values as long as they are `constexpr`.

```
enum Color : int { red = 0, green = 1, blue = 2 };
```

Once again, the guidelines recommends us not to do this⁵, unless it's necessary! Once you start doing it, it's easy to make mistakes and mess it up. We can rely on the compiler assigning subsequent values to the subsequent enumerator values.

- A good reason to specify the enumerator value is to define only the starting value. If you define the months and you don't want to start with zero.

```
enum Month { jan = 1, feb, mar, apr, may, jun,
    jul, august, sep, oct, nov, dec };
```

- Another reason can be if you want to define the values as some meaningful character

```
enum altitude: char {
    high = 'h',
    low = 'l'
};
```

⁴<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#enum7-specify-the-underlying-type-of-an-enumeration-only-when-necessary>

⁵<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#enum8-specify-enumerator-values-only-when-necessary>

- One other reason can be emulating some bitfields. So you don’t want subsequent values, but you always want the next power of two

```
enum access_type { read = 1, write = 2, exec = 4 };
```

29.2 Scoped enumerations

In the previous section, we saw declarations such as `enum EnumName{};`. C++11 brought a new type of enumeration called scoped `enums`. They declared either with the `class` or with the `struct` keywords and there is no difference between those two.

The syntax is the following:

```
enum class Color { red, green, blue };
```

For scoped enumerations the default underlying type is defined in the standard, it is `int`. This also means that if you want to forward declare a scoped `enum`, you don’t have to specify the underlying type. If it is meant to be `int`, this is enough:

```
enum class Color;
```

Apart from how the syntactical differences between how they are declared, what other differences exist?

Unscoped `enums` can be implicitly converted to their underlying type. Implicit conversions are often not what you want, and scoped `enums` don’t have this “feature”. Exactly because of the unwelcome implicit conversions, the Core Guidelines strongly recommends using scoped over unscoped `enums`.

```
1 void Print_color(int color);
2
3 enum Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
4 enum Product_info { red = 0, purple = 1, blue = 2 };
5
6 Web_color webby = Web_color::blue;
7
8 // Clearly at least one of these calls is buggy.
9 Print_color(webby);
10 Print_color(Product_info::blue);
```

Unscoped `enums` export their enumerators to the enclosing scope which might lead to name clashes. On the other hand, with scoped `enums`, you must always specify the name of the `enum` alongside with the enumerators.

```
1 enum UnscopedColor { red, green, blue };
2 enum class ScopedColor { red, green, blue };
3
4 int main() {
5     [[maybe_unused]] UnscopedColor uc = red;
6     // [[maybe_unused]] ScopedColor sc = red; // Doesn't compile
7     [[maybe_unused]] ScopedColor sc = ScopedColor::red;
8 }
```

29.3 What else

Now that we saw how un/scoped **enums** work and what are the differences between them, let's see what other **enum** related functionalities the language or standard library offers.

`std::is_enum`

C++11 introduced the `<type_traits>` header. It includes utilities to check the properties of types. Not surprisingly `is_enum` is there to check whether a type is an **enum** or not. It returns `true` both for scoped and unscoped versions.

Since C++17, `is_enum_v` is also available for easier usage.

```

1 #include <iostream>
2 #include <type_traits>
3
4 enum UnscopedColor { red, green, blue };
5 enum class ScopedColor { red, green, blue };
6 struct S{};
7
8 int main() {
9     std::cout << std::boolalpha
10    << std::is_enum<UnscopedColor>::value << '\n'
11    << std::is_enum<ScopedColor>::value << '\n'
12    << std::is_enum_v<S> << '\n';
13 }
```

`std::underlying_type`

`std::underlying_type` was also an addition to C++11. It helps us retrieve the underlying type of an **enum**. Until C++20 if the checked **enum** is not completely defined or not an **enum**, the behaviour is undefined. Starting with C++, the program becomes ill-formed for incomplete **enum** types.

C++14 introduced a related helper, `std::underlying_type_t`.

```

1 #include <iostream>
2 #include <type_traits>
3
4 enum UnscopedColor { red, green, blue };
5 enum class ScopedColor { red, green, blue };
6 enum class CharBasedColor : char { red = 'r', green = 'g', blue = 'b' };
7
8 int main() {
9
10    constexpr bool isUnscopedColorInt =
11        std::is_same_v< std::underlying_type<UnscopedColor>::type, int >;
12    constexpr bool isScopedColorInt =
13        std::is_same_v< std::underlying_type_t<ScopedColor>, int >;
14    constexpr bool isCharBasedColorInt =
```

```

15     std::is_same_v< std::underlying_type_t<CharBasedColor>, int >;
16     constexpr bool isCharBasedColorChar =
17         std::is_same_v< std::underlying_type_t<CharBasedColor>, char >;
18
19     std::cout
20         << "underlying type for 'UnscopedColor' is "
21             << (isUnscopedColorInt ? "int" : "non-int") << '\n'
22         << "underlying type for 'ScopedColor' is "
23             << (isScopedColorInt ? "int" : "non-int") << '\n'
24         << "underlying type for 'CharBasedColor' is "
25             << (isCharBasedColorInt ? "int" : "non-int") << '\n'
26         << "underlying type for 'CharBasedColor' is "
27             << (isCharBasedColorChar ? "char" : "non-char") << '\n'
28     ;
29 }
```

29.4 Using-enum-declaration since C++20

Since C++20, we can use `using` with `enums`. It introduces the enumerator names in the given scope.

The feature is smart enough to raise a compilation error in case a second `using` would introduce an enumerator name that was already introduced from another `enum`.

```

1 #include <type_traits>
2
3 enum class ScopedColor { red, green, blue };
4 enum class CharBasedColor : char { red = 'r', green = 'g', blue = 'b' };
5
6 int main() {
7     // OK!
8     using enum ScopedColor;
9     // error: 'CharBasedColor CharBasedColor::red' conflicts with a previous declaration
10    using enum CharBasedColor;
11 }
```

It's worth noting that it doesn't recognize if an unscoped enum already introduced an enumerator name in the given namespace. In the following example, there is already `red`, `green`, and `blue` available from `UnscopedColor`, still, the `using` of `ScopedColor` with the same enumerator names is accepted.

```

1 #include <type_traits>
2
3 enum UnscopedColor { red, green, blue };
4 enum class ScopedColor { red, green, blue };
5
6 int main() {
```

```

7     using enum ScopedColor;
8 }
```

29.5 C++23 brings std::is_scoped_enum

C++23 will introduce one more `enum` related function in the `<type_traits>` header, one of it is `std::is_scoped_enum` and it's helper function `std::is_scoped_enum_v`. As the name suggests and the below snippet proves, it checks whether its argument is a **scoped enum** or not.

```

1 #include <iostream>
2 #include <type_traits>
3
4 enum UnscopedColor { red, green, blue };
5 enum class ScopedColor { red, green, blue };
6 struct S{};
7
8 int main()
9 {
10     std::cout << std::boolalpha;
11     std::cout << std::is_scoped_enum<UnscopedColor>::value << '\n';
12     std::cout << std::is_scoped_enum_v<ScopedColor> << '\n';
13     std::cout << std::is_scoped_enum_v<S> << '\n';
14     std::cout << std::is_scoped_enum_v<int> << '\n';
15 }
16 /*
17 false
18 true
19 false
20 false
21 */
```

If you want to try out C++23 features, use the `-std=c++2b` compiler flag.

29.6 C++23 introduces std::to_underlying

C++23 will introduce another library feature for `enums`. The `<utility>` header will be enriched with `std::to_underlying`. It converts an `enum` to its underlying type. As mentioned, this is a library feature, meaning that it can be implemented in earlier versions.

This one can be replaced with a `static_cast` if you have access only to earlier versions: `static_cast<std::underlying_type_t<MyEnum>>(e);`.

```

1 #include <iostream>
2 #include <type_traits>
3 #include <utility>
```

```
4
5 enum class ScopedColor { red, green, blue };
6
7 int main()
8 {
9     ScopedColor sc = ScopedColor::red;
10    [[maybe_unused]] int underlying = std::to_underlying(sc);
11    [[maybe_unused]] int underlyingEmulated
12        = static_cast<std::underlying_type_t<ScopedColor>>(sc);
13    [[maybe_unused]] std::underlying_type_t<ScopedColor> underlyingDeduced
14        = std::to_underlying(sc);
15 }
```

As a reminder, let me reiterate that if you want to try out C++23 features, use the `-std=c++23` compiler flag.

29.7 Conclusion

In this article, we discussed all the language and library features that are about enumerations. We saw how scoped and unscoped `enums` differ and why it's better to use scoped `enums`. That's not the only Core Guidelines recommendation we discussed.

Then we checked how the standard library has been enriched during the years supporting an easier work with `enums`. We also had a sneak peek into the future and checked what C++23 will bring for us.

Part IV

Metaprogramming

C++ 中谈论元编程，一般是指编译期的编程，分为三个时期。

第一个时期，模板时期，以模板作为编译期计算工具。这是自上世纪九十年代就在 C++ 中开始应用的特性，借其得以在 C++ 中实现泛型编程，一堆奇技淫巧也随之产生，这些“太过聪明”的技巧，需要对语言有深入的理解，才能够设计并维护这种代码。

第二个时期，常量表达式时期，以编译期表达式特性作为编译期计算工具。为简化必须使用模板编写编译期代码的方式，从 C++11 开始陆续引入了一系列编译期特性，例如 *constexpr*、*constexpr if*、*constinit/consteval* 和 C++23 *consteval if* 等等。常量表达式为编译期操纵“值”提供了基础，而模板元编程为涉及“类型”的逻辑计算提供了工具，这无疑丰富了元编程的工具。

第三个时期，静态反射时期。模板与常量表达式是元编程系统的基础，而静态反射是元编程系统的核心，它带来的是一种产生式元编程能力。这将使 C++ 步入一个全新的时期，一个完善、灵活、强大的元编程时期。

关于第二个时期的元编程，相关文章被归类到了 **Performance** 这一 Part。本 Part 的文章则集中于第三时期元编程，包含四篇，围绕 C++ 反射进行了详细讨论，是目前网上少有的全面讲解 C++ 静态反射的文章。相信会使你对 C++ 元编程有更加全面的理解。

2023 年 5 月 13 日
里缪

Chapter 30

C++ 反射 第一章 通识

• 里缪 2022-02-18

本篇开始介绍 C++ 反射，这是第一篇。

30.1 C++ 中的产生式元编程

元编程在 C++ 已有几十年的历史，一般来说，我们是指编译期的编程。

产生式元编程，则侧重于强调「产生代码的代码」这种编程方式。

C++ 提供了许多生成代码的特性，如图30.1所示。

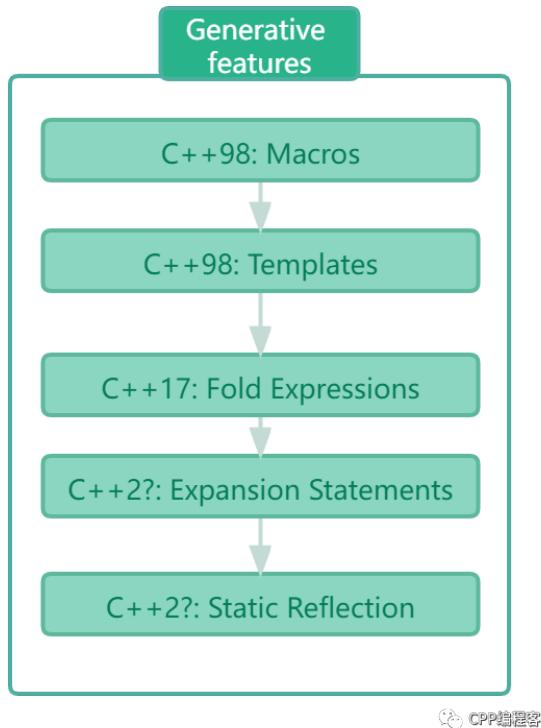


图 30.1: Generative Features

宏虽然只是 C 中的一个简单替换的工具，却也具有强大的产生代码的能力。过去的文章多次使

用它来简化重复的工作，想必大家并不陌生。

模板作为 C++ 元编程的工具，可以完成泛型编程「抽象化」的工作，借其可实现变量、函数、类的泛化，从而自动生成百上千行代码。同样，大家对此自然更加熟悉。

Fold Expressions 是 C++17 提供的展开参数包的简便方式，这同样是一种产生代码的特性。见【C++17: Simplify Code with Fold Expressions】¹。

Expansion Statements 是 P1306 提出的一种新语句，可以减少遍历时的重复，主要是方便反射遍历用的。这也是一种产生代码的特性，举个遍历 tuple 的例子，

```
auto tup = std::make_tuple(0, 'a', 3.14);
template for (auto& elem : tup)
    std::cout << elem << std::endl;
```

语法起初是 for...，后来改为了 template for，这代码相当于如下代码：

```
1 auto tup = std::make_tuple(0, 'a', 3.14);
2 {
3     auto elem = std::get<0>(tup);
4     std::cout << elem << std::endl;
5 }
6 {
7     auto elem = std::get<1>(tup);
8     std::cout << elem << std::endl;
9 }
10 {
11     auto elem = std::get<2>(tup);
12     std::cout << elem << std::endl;
13 }
```

该提案由于时间原因没能进入 C++20，之后三年没动静，最近 CWG 进行了 review，但是作者一直没有更新论文，也没能进入 C++23。似乎放弃了？²

虽然已经拥有这么多生成代码的特性，但是我们依旧无法轻易完成像序列化、ORM、远程调用、schema generation 等等需求。因为 C++ 缺少获取类型元信息的机制，所以无法获取像类型的参数列表、成员类型、成员名称等等这些信息。

缺少的就是反射的机制，这是一种产生代码更加强大的能力。

C++ 早已成立了专门的小组 SG7 来负责反射的研究工作，近些年也算是取得了一些发展，最快也许 C++26 可以加入。

30.2 反射的相关概念

本节简单地说下反射相关的概念，以建立共识。

首先给大家介绍两个词：reflection 和 reification。

反射一般包含两个部分，获取和构建，如图30.2所示：

获取指的是从类型得到「类型元信息」，元信息要比类型高一层，所以是「自下而上」的结构。也就是说，这是一种从具体到抽象的结构，这个步骤就称为 reflection。

¹<https://tinyurl.com/cppmore-fold-expression>

²<https://github.com/cplusplus/papers/issues/156>

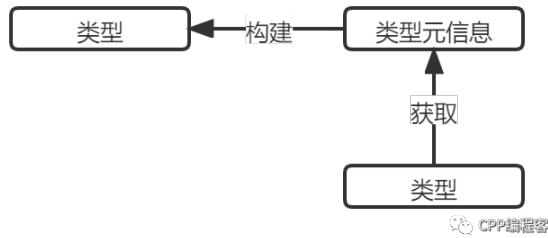


图 30.2: Reflection 框架

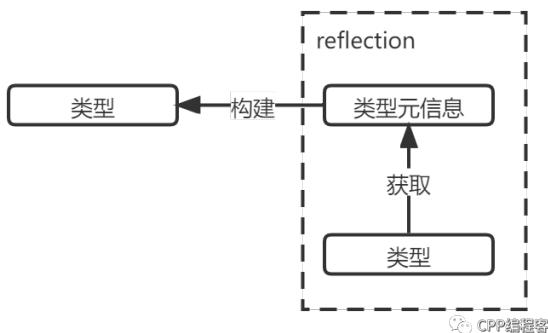


图 30.3: Reflection

而构建指的是从「类型元信息」再次得到类型，这是「自上而下」的结构，因此它是从抽象到具体，这个完全相反的步骤就称为 **reification**。

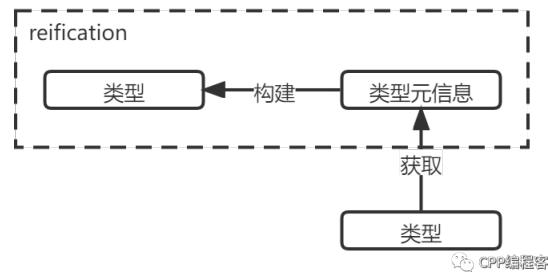


图 30.4: Reification

接着，再向大家介绍一个词：**Introspection**。

简而言之，这指的是询问某个类型是否具有什么东西的特性。比如，查询一个类型是否派生自另一个类型，是否具有 `get_data()` 成员函数，是否拥有 `data` 属性，是否能转换成另一个类型，诸如此类。

这个特性就是反射能力的一部分，其实 C++ 通过 `type traits` 和 `Concepts` 已经提供许多相关能力了。

C++ 缺少的是类型遍历的能力，有了这种能力，就能够遍历出类的模板参数、成员类型和函数列表这些信息，再通过 Introspection 便可操纵某些具体的变量或函数，自动产生其他版本的类。

到此，本节就先介绍这几个比较大的概念。

30.3 C++ 中的反射

反射分为动态反射和静态反射，动态反射就是运行期的反射，静态反射就是编译期的反射。

C++ 的反射是静态反射，第一个 Reflection TS 基于 N4766。当时还是基于类型来表示反射信息，举个例子：

```

1 template <typename T>
2 std::string get_type_name() {
3     namespace reflect = std::experimental::reflect;
4     // T_t is an Alias reflecting T:
5     using T_t = reflexpr(T);
6     // aliased_T_t is a Type reflecting the type for which T is a synonym:
7     using aliased_T_t = reflect::get_aliased_t<T_t>;
8     return reflect::get_name_v<aliased_T_t>;
9 }
10 std::cout << get_type_name<std::string>(); // outputs basic_string

```

看不懂也没关系，因为早就废弃这种方式了。

这种方式是为了简化和模板元编程的结合，然而出于多方面考虑，SG7 转而支持 value-based reflection，也就是现在的反射。为此，C++20 提供了许多扩展特性来支持反射的设计，例如 consteval function，`std::is_constant_evaluated()`，`constexpr dynamic allocation`。

新式的反射语法，举个例子：

```

1 #include <meta>
2 template<Enum T>
3 std::string to_string(T value) {
4     template for (constexpr auto e : std::meta::members_of(^T)) {
5         if([:e:] == value) {
6             return std::string(std::meta::name_of(e));
7         }
8     }
9     return "<unnamed>";
10 }

```

这段代码是要以 string 形式输出枚举类型的值。

获取类型元信息的操作符是“^ operator”，读作 **lifting operator**，意思就是向上获取类型的元信息。这对应上一节介绍的 **reflection** 这个词。³

通过 `std::meta::members_of()` 便可通过类型元信息获取到枚举类型的所有成员，以 `std::span` 返回。

如何遍历返回的结果呢？就用到了前面介绍的 expansion statements，也就是代码中的 `template for`。

那么如何再把类型 reification 出来呢？语法就是 `[: reflection :]`，这个称为 **splice construct**。C++ 把这个过程称为 **splicing**，它和 reification 指的是一个东西。

通过比较枚举值，相同则使用 `std::meta::name_of()` 返回枚举值的名称，以 `std::string_view` 返回。

³ reflection 操作的原本语法是 `reflexpr()`，太重已被更换。

上面的例子很简洁的说明了 C++ 反射的用法，贯穿了上节介绍的概念，可以让大家先对反射有个大体的理解。

30.4 百花齐放

C++ 反射进入标准的速度如此慢，导致在过去这么多年，已经产生了许多用户自己实现反射的办法。

这大体可以分为三个阶段，如图30.5所示。

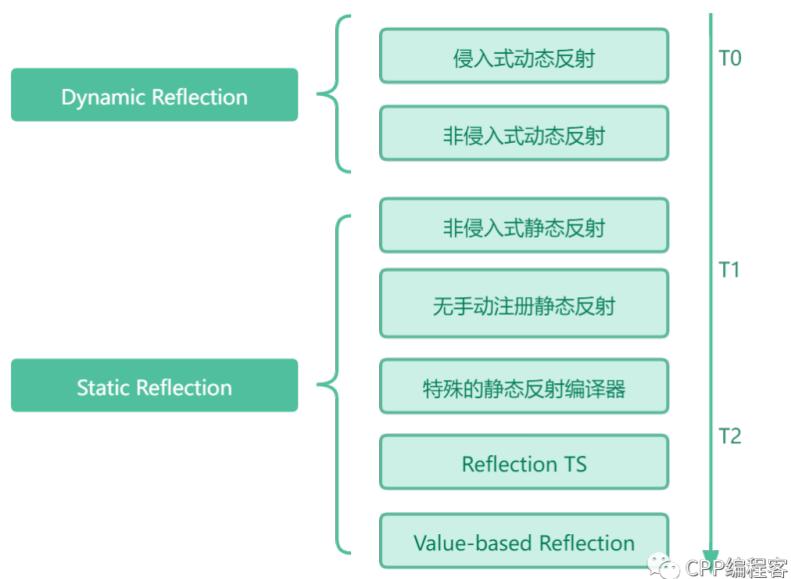


图 30.5: 反射实现手法

首先来说 T0 阶段。

因为没有类型元信息机制，这些实作手法往往需要自己「保存」类型信息，然后再根据保存的内容「获取」相关的反射信息。

这些实现手法在编译期完成这些「保存」工作，就属于静态反射，反之则是动态反射。

不论如何，这些实现使用起来都有不少束缚，比如需要用户手动注册类型信息、不支持有些 Introspection、操作繁琐等等。

其次来说 T1 阶段，这些实现不需要用户手动注册类型信息。

第一种方式是采用 TMP 的一些技巧，保存类型的一些信息，使用起来比较方便，但局限不小，比如无法获取成员的名字。

第二种方式比较高大上，它通过提供特殊的编译器来提供类型元信息，也就是自己实现一套内置的反射系统。这种方式的反射特性比较完整，使用起来很方便，功能很强大，但是这些扩展换个编译器就无法编译，相当于是没有达成标准的野路子。

最后是 T2 阶段，也就是 C++ 标准的反射。

前面已经介绍过，故不再赘述。需要注意的是，这里 Reflection TS 强调的是目前编译期支持的标准反射，而 Value-based reflection 强调的是最新的反射。

30.5 总结

如题所言，本篇是讲反射「通识」。

因此不会太涉及细节，目的是先从大的方向上让大家对 C++ 反射有个整体的了解。

后面的章节，则会再来针对每个模块谈论细节。

原本只打算写一篇的，包含一些细节，但是内容比想象的要多，所以拆解成为更有层次的一个小系列，不会太多，四五篇足矣。

Chapter 31

C++ 反射 第二章 探索

• 里缪 曲 2022-02-26 🎵★★

继续更新第二篇，本文将介绍一些 T0 和 T1 阶段的 C++ 反射库，对比探索其差异。

31.1 动态反射 & 静态反射

动态反射，就是类型元信息在运行期产生的反射，而静态反射则是在编译期产生。

诸多反射库，都借助了各种办法在运行期或是编译期来保存类型的信息，从而实现反射的能力。这些实现又有「侵入式」和「非侵入式」之分。

侵入式会在你的类中添加一些保存类型信息的组件，这往往会采用宏来自动生成，隐藏实现细节。

这种方式可以得到类型中的私有信息，因为往往会被保存信息的组件置为友元。但毫无疑问，此法会破坏原有类型的结构，一般这种库没人会用。

非侵入式则是在类外声明一些组件来收集类型元信息，这也是大多数反射库采取的实现方式。

同样，具体实现也会被隐藏起来，使用宏来自动生成相关代码，主要原因是为了使用方便。然而，此法无法得到类型中的私有信息，这是一处限制。另外，有些库利用一些技巧，从而无需手动注册类型信息，不过限制亦多。

总的来说，不可避免地，大多数库都需要用户手动地填写类型信息，这是反射机制缺少所致，因此使用起来多少都会有些限制。

以下各节，分别来介绍一些 C++ 的反射库，探索一下基本用法。

31.2 Boost Describe Library

第一，来看 Boost 中包含的一个 C++14 反射库：Describe。

该库支持枚举、结构体和类的反射，先来看个官方提供的小例子：

```
1 #include <iostream>
2 #include <boost/describe.hpp>
3 #include <boost/mp11.hpp>
4
5 enum E
```

```

6   {
7     v1 = 1,
8     v2 = 2,
9     v3 = 3
10    };
11
12 BOOST_DESCRIBE_ENUM(E, v1, v2, v3)
13
14 int main()
15 {
16   using L1 = boost::describe::describe_enumerators<E>;
17
18   boost::mp11::mp_for_each<L1>([](auto D) {
19
20     std::cout << D.name << ":" << static_cast<int>(D.value) << std::endl;
21
22 });
23
24
25   return 0;
26 }

```

这个例子同样是用于输出枚举类型的值。

那么它是从哪得到枚举类型信息的呢？注意这里的 BOOST_DESCRIBE_ENUM 宏，此宏便是用来自动产生收集「枚举类型的元信息」的代码。

这个宏支持可变参数，第一个参数就是枚举类型，其后的参数则是枚举中的字段。

通过这种手动注册的方式，该库就可以提供类型的元信息。

对于以上例子，这个宏将自动展开成如下代码：

```

1 static_assert(std::is_enum < E > ::value,
2               "BOOST_DESCRIBE_ENUM should only be used with enums");
3 inline auto boost_enum_descriptor_fn(E*) {
4   return boost::describe::detail::enum_descriptor_fn_impl(0, [] {
5     struct _boost_desc {
6       static constexpr auto value() noexcept {
7         return E::v1;
8       }
9       static constexpr auto name() noexcept {
10        return "v1";
11      }
12    };
13    return _boost_desc();
14  }(), [] {
15    struct _boost_desc {

```

```

16         static constexpr auto value() noexcept {
17             return E::v2;
18         }
19         static constexpr auto name() noexcept {
20             return "v2";
21         }
22     };
23     return _boost_desc();
24 } (), [] {
25     struct _boost_desc {
26         static constexpr auto value() noexcept {
27             return E::v3;
28         }
29         static constexpr auto name() noexcept {
30             return "v3";
31         }
32     };
33     return _boost_desc();
34 } ());
35 }
```

可以看到，这里自动产生了一系列代码来保存枚举类型的信息。

根据自动产生的这个 `boost_enum_descriptor_fn` 函数，它就能构建类型的描述信息，

```
template<class E> using describe_enumerators =
    decltype( boost_enum_descriptor_fn( static_cast<E*>(0) ) );
```

也因此，主函数中的 `describe_enumerators` 才有了定义：

```

1 using L1 = boost::describe::describe_enumerators<E>;
2
3 boost::mp11::mp_for_each<L1>([](auto D) {
4
5     std::cout << D.name << ":" << static_cast<int>(D.value) << std::endl;
6
7});
```

然后，通过 `mp11` 的 `mp_for_each` 便可以迭代类型的所有信息。

该库使用起来还算方便，但是也有一些限制，后面再来看。

31.3 RTTR Library

第二，来看一个使用人数较多的 C++ 动态反射库：rttr。

该库的文档比较充足，有专门的网站 www.rttr.org，因此遇到问题的话比较好解决。

同样，先来看一个简单的例子：

```

1 #include <rttr/registration>
2 #include <iostream>
3 using namespace rttr;
4
5
6 struct Point {
7     int x;
8     int y;
9     void print() {}
10 };
11
12 RTTR_REGISTRATION
13 {
14     registration::class_<Point>("Point")
15         .constructor<>()
16         .property("x", &Point::x)
17         .property("y", &Point::y)
18         .method("print", &Point::print);
19 }
20
21 int main()
22 {
23     type t = type::get<Point>();
24     for (auto& prop : t.get_properties())
25         std::cout << "name: " << prop.get_name() << std::endl;
26
27     return 0;
28 }
```

这个例子表示了如何使用 rttr 库来获取 struct 的类型信息。

它又是如何收集类型信息的呢？

注意其中的 RTTR_REGISTRATION 宏，这个宏用于产生初始化「注册类型元信息」的代码，注册类型元信息的代码应该写在该宏的下方，通过 registration::class_ 来添加你要保存的类型信息。

若将代码展开，则相当于：

```

1 static void rttr_auto_register_reflection_function_();
2 namespace {
3     struct rttr__auto__register__ {
4         rttr__auto__register__() {
5             rttr_auto_register_reflection_function_();
6         }
7     }
8     ;
9 }
```

```

10 static const rttr::auto_register__ auto_register__12;
11 static void rttr_auto_register_reflection_function_()
12 {
13     registration::class_<Point>("Point")
14         .constructor<>()
15         .property("x", &Point::x)
16         .property("y", &Point::y)
17         .method("print", &Point::print);
18 }

```

这里自动生成了 `rttr_auto_register_reflection_function_()` 函数的声明，用于注册反射信息，宏下方所写的注册类型元信息的代码其实就是该函数的定义。

通过定义一个静态 `rttr::auto_register_` 对象，在其构造函数中调用该注册函数，从而完成初始化类型元信息的收集工作。

之后，便可通过 `type t = type::get<Point>();` 获取保存的类型元信息，获取想要的属性。

该库注册类型信息稍显麻烦，使用倒是比较简单。

31.4 Cista Library

第三，介绍一个 C++17 序列化库：Cista，它也提供一些反射能力。

该库只有一个头文件，直接拷贝到项目中就能用，非常方便。它也有专门的网站 <https://cista.rocks/>。

来看一个简单的例子：

```

1 #include <iostream>
2 #include "cista.h"
3
4
5 struct a {
6     int i_ = 1;
7     int j_ = 2;
8     double d_ = 100.0;
9     std::string s_ = "hello";
10};
11
12 int main() {
13     a i;
14     cista::for_each_field(
15         i, [] (auto&& m) {
16             std::cout << m << std::endl;
17 });
18
19
20     return 0;
21 }

```

咦！这里怎么不需要注册类型信息呢？

它的实现借助了 structured bindings，把 struct 的所有成员组成了一个 tuple，再从 tuple 来遍历出所有的成员。

由于使用了这种技术，所以它只能得到成员的值，而无法得到成员的名称。

这种实作法源自 `magic_get`¹，该库提供了一种著名的无手动注册反射实现手法，CppCon 2016 的演讲²专门介绍了该实作法。

该库主要支持数据序列化功能，看个官方的简洁例子：

```

1 #include <cassert>
2 #include <iostream>
3 #include "cista.h"
4
5 int main() {
6     namespace data = cista::raw;
7     struct my_struct { // Define your struct.
8         int a_{ 0 };
9         struct inner {
10             data::string b_;
11         } j;
12     };
13
14     std::vector<unsigned char> buf;
15     { // Serialize.
16         my_struct obj{ 1, {data::string{"test"}} };
17         buf = cista::serialize(obj);
18     }
19
20     // Deserialize.
21     auto deserialized = cista::deserialize<my_struct>(buf);
22     assert(deserialized->j.b_ == data::string{ "test" });
23
24     return 0;
25 }
```

由于不需要手动注册类型信息，你可以轻松将类型的数据保存起来，再进行恢复，性能很高。

总而言之，这个库设计所用到的技巧比较精妙，跟其他反射库的设计思路完全不一样。代价就是反射能力不足，比如无法获取类型和成员的名字。

31.5 iguana Library

最后，介绍下国人开发的一个 C++17 序列化库：iguana³，其中包含有静态反射。

¹https://github.com/apolukhin/magic_get

²标题为“C++14 Reflections Without Macros, Markup nor External Tooling.”

³<https://github.com/qicosmos/iguana>

使用该库，可以轻松将对象序列化为 xml, json 等常用形式，举个自带的小例子：

```

1 #include <iguana/json.hpp>
2
3 struct person
4 {
5     std::string name;
6     int age;
7 };
8
9 REFLECTION(person, name, age) //define meta data
10
11
12 int main()
13 {
14     person p = { "tom", 28 };
15
16     iguana::string_stream ss;
17     iguana::json::to_json(ss, p);
18
19     std::cout << ss.str() << std::endl;
20
21     return 0;
22 }
```

这将把 person 映射成 json 格式：

```
{"name": "tom", "age": 28}
```

这里是通过 REFLECTION 宏来自动产生保存类型信息的代码，上述代码中将展开为：

```

1 constexpr inline std::array<std::string_view, 2> arr_person = {
2     std::string_view("name", sizeof("name") - 1), std::string_view("age", sizeof("age") - 1)
3 }
4 ;
5 static auto iguana_reflect_members(person const&) {
6     struct reflect_members {
7         constexpr decltype(auto) static apply_impl() {
8             return std::make_tuple(&person::name, &person::age);
9         }
10        using type = void;
11        using size_type = std::integral_constant<size_t, 2>;
12        constexpr static std::string_view name() {
13             return std::string_view("person", sizeof("person") - 1);
14         }
15        constexpr static size_t value() {
```

```

16     return size_type::value;
17 }
18 constexpr static std::array<std::string_view, size_type::value> arr() {
19     return arr_person;
20 }
21 };
22
23 return reflect_members{};
24 }
```

所有的类型信息将保存到 reflect_members 之中，通过如下代码就能得到类型的反射信息：

```
using M = decltype(iguana_reflect_members(std::forward<T>(t)));
```

reflect_members 中的 name() 保存了类型的名称，apply_impl() 通过构建 tuple 保存了成员的类型，arr() 则保存了每个成员的名称，size_type 保存了成员的数量。

关于反射的使用例子请看下节。

31.6 Schema Generation

上面几节介绍了几个反射库的元信息产生手法和基本用法，作为对比，本节使用上述库来实现同一个功能，大家可以感受下其差异与便捷程度。

我们将使用这些库提供的反射能力，来为一个类型自动生成 SQL 语句，测试的结构体如下：

```
struct Account {
    int id{ 100 };
    std::string name{"jack"};
};
```

第一个，来看看如何使用 Boost Describe 实现该任务。

首先使用宏来注册类型信息，代码很简单：

```
BOOST_DESCRIBE_STRUCT(Account, (), (id, name))
```

注册类的宏需要提供三个参数，第一个是类的名称，第二个是继承类列表，第三个是属性列表，包含成员变量和成员函数。

接着，创建产生数据表的函数，代码如下：

```

1 template<typename T,
2         typename Md = boost::describe::describe_members<T, boost::describe::mod_any_access>>
3 std::string create_table() {
4     std::stringstream result;
5     int num = __member_number(T{});
6     result << "CREATE TABLE " << __type_name(T{}) << "(\n";
7
8     boost::mp11::mp_for_each<Md>([&](auto D) {
9         // create column
```

```

10     result << D.name << " ";
11     result << to_sql<decltype(T{*}.*D.pointer)>();
12
13     if (--num != 0)
14         result << ",\n";
15 };
16 result << ");\n";
17 return result.str();
18 }

```

模板参数 T 就是欲创建 SQL 语句的类型，通过 describe_members 来得到该类型的所有成员信息，以 Md 表示。

在函数内部，开始组装 SQL 语句，此时创建表需要类型的字符式名称，然而 Describe 没有提供这个信息。因此，为了简便起见，我们直接修改源码，添加这两个反射能力，修改代码如下：

```

1 /* 此处修改了源码，方便获取类型名称和成员数量 */
2 #include <boost/preprocessor/variadic/size.hpp>
3 #define BOOST_DESCRIBE_STRUCT(C, Bases, Members) \
4     inline constexpr char const* __type_name(C const&) { return #C; } \
5     inline constexpr int __member_number(C const&) { \
6         return BOOST_PP_VARIADIC_SIZE(BOOST_DESCRIBE_PP_UNPACK Members); } \
7     static_assert(std::is_class<C>::value, "BOOST_DESCRIBE_STRUCT should only be used with class" \
8 BOOST_DESCRIBE_BASES(C, BOOST_DESCRIBE_PP_UNPACK Bases) \
9 BOOST_DESCRIBE_PUBLIC_MEMBERS(C, BOOST_DESCRIBE_PP_UNPACK Members) \
10 BOOST_DESCRIBE_PROTECTED_MEMBERS(C) \
11 BOOST_DESCRIBE_PRIVATE_MEMBERS(C)

```

同样，它也没有提供直接获取成员个数的能力，因而此处添加了 __member_number() 来提供该能力，使用 __type_name() 则可以得到类型的名称。

之后，便可以开始为每个成员创建 column，类型使用 to_sql() 来处理，代码同样简单：

```

1 template<typename T>
2 const char* to_sql() {
3     static_assert(false, "no translation to SQL");
4 }
5
6 template<>
7 const char* to_sql<int>() {
8     return "INTEGER";
9 }
10
11 template<>
12 const char* to_sql<std::string>() {
13     return "TEXT";
14 }

```

通过为指定类型提供 `to_sql` 特化版本，就能返回相应类型的 SQL 类型。

可以调用 `create_table` 看看效果：

```

1 std::cout << create_table<Account>();
2
3 // output:
4 CREATE TABLE Account(
5   id INTEGER,
6   name TEXT);

```

`Describe` 实现这个功能如此简单，是因为我们修改了源码，增加了一些反射能力。其本身若只用来处理每个成员，则比较简单。

第二个，来看 `rttr` 如何完成该任务。

同样，首先注册类型信息，

```

1 RTTR_REGISTRATION
2 {
3     rttr::registration::class_<Account>("Account")
4         .constructor<>()
5         .property("id", &Account::id)
6         .property("name", &Account::name);
7 }

```

相较而言，`rttr` 的这种实现注册信息稍微麻烦，需要手动填写太多内容。

接着，同样来编写相关函数，代码如下：

```

1 template<typename T>
2 auto create_table() {
3     rttr::type t = rttr::type::get<Account>();
4     int num = t.get_properties().size();
5     std::stringstream result;
6     result << "CREATE TABLE " << t.get_name() << "(\n";
7
8     for (auto& e : t.get_properties()) {
9         result << e.get_name() << " ";
10        auto type_name = e.get_type().get_name().to_string();
11        result << to_sql(type_name);
12        if (--num != 0)
13            result << ",\n";
14    }
15
16    result << ");\n";
17    return result.str();
18 }

```

`rttr` 的反射能力提供的比较完整，所有需要的信息都能直接得到，命名亦是一目了然，使用起来体验还不错。

代码逻辑比较清晰，就不过多赘述，主要来看下 `to_sql()`。

`Describe` 的实现使用了偏特化来处理类型，而 `rttr` 对属性使用了类型擦除，所以想要直接得到成员类型反而困难。不过获取字符类型的比较简单，因此 `to_sql()` 也转换实现方案，实现如下：

```

1 const char* to_sql(const std::string& type_name) {
2     static std::unordered_map<std::string, const char*> types{
3         {"int", "INTEGER"}, 
4         {"std::string", "TEXT"} 
5     };
6     auto p = types.find(type_name);
7     if (p == types.end())
8         return "";
9
10    return p->second;
11 }
```

这里采用映射的方案来处理类型，为简单起见，不支持的类型直接返回为空。

调用这个 `rttr` 的实现将会得到相同的结果。

最后一个，来看 `iguana` 如何完成这个任务。

为何跳过 `cista` 呢？因为它的实现和其它反射库的思路不一样，虽然不用手动注册类型信息，却也没办法获取成员的名称。

使用 `iguana` 来注册类型信息，代码如下：

```
REFLECTION(Account, id, name)
```

相对来说，手动填写的信息很少，但是也有舍弃，只能支持成员变量，函数、继承等等都不支持。

接着继续实现 `create_table` 函数，代码如下：

```

1 template<typename T>
2 constexpr auto create_table(T&& t) {
3     using M = decltype(iguana_reflect_members(std::forward<T>(t)));
4     int num = 0;
5
6     std::stringstream result;
7     result << "CREATE TABLE " << M::name() << "(\n";
8     iguana::for_each(std::forward<T>(t), [&num, &result, &t](auto &v, auto i) {
9         auto name = iguana::get_name<decltype(t), decltype(i)::value>();
10        result << name << " ";
11        result << to_sql<std::decay_t<decltype(t.*v)>>();
12        if (++num != M::value())
13            result << ",\n";
14    });
15    result << ");\n";
16    return result.str();
17 }
```

实现起来也相比较简单，但是只有阅读了源码，你才能清楚如何获得想要的信息。

这毕竟是个序列化库，只是提供了转换成 xml, json 等格式的简便接口。反射属于幕后角色，不熟悉实现的话用起来比较麻烦。

代码逻辑与前面相同，故亦不赘述，调用该实现将得到相同的结果。

31.7 总结

限于篇幅，本文只是选取了几个实现有所差异的 C++ 反射库进行讨论。

还有许多库，例如 ponder、refl-cpp、CPP-Reflection 等等，若感兴趣可以去看看。

若追求稳定，那选择 rttr 没问题，它的文档相当多，源码当中包含有规范的注释，反射能力相对完善，这也是使用人数较多的原因。

若追求效率，iguana 和 refl-cpp 支持静态反射，前者文本已经介绍，后者用法有些繁琐，用哪个须得自己斟酌。

若不需要获取字段名称，cista 这种无需手动注册的反射库也比较好用，序列化效率很高。

总之，这些反射库的实现都有取舍，需要根据实际需求进行选择。

下篇再带大家看一下完备反射所具有的能力。

Chapter 32

C++ 反射 第三章 原生

• 里缪 曲 2022-03-11 📅 ★★★

工具能够提升开发效率，却也会限制住思维。

缺少语言机制而实现的反射，就像是停电之时点着蜡烛来取光，虽够用，然多是不便。

原生反射的机制位于原理层，属于底层的、自动的机制，无需用户手动注册任何类型元信息。同时，可以获取类型的 *Introspection* 信息比较完整，使用起来非常灵活，在此之上，可以轻松构建许多强大的组件。

上一章，介绍了诸多不同实现的自定义反射库，丰富了开发工具。但是，这些反射库仅属于结构层，反射能力并不完备，取光可以，代替整个反射概念却有不及。

若是仅会使用其中的几个库，对反射在应用层面的思索不免受到桎梏。

因此，本篇介绍一个原生支持静态反射的 C++ 编译器：Circle¹。

Sean Baxter 对 C++ 进行了一系列扩展，增加了许多现代语言所具有的特性，便有了这个新式的 C++20 编译器。扩展的特性包含 *dynamic pack property*, *universal member access*, *constexpr conditional*, *imperative arguments* 等等现代特性，此外，作者也实现了一些 C++23 提案的扩展，例如 *pattern matching*, *if consteval*, *deducing this*。

静态反射，是其中非常重要的一个扩展，是很多扩展的基石。

32.1 基本 Introspection

上篇当中，展示了诸多 T0 层反射库的使用例子，其用法难免繁琐，能力也很弱小，与 Circle 原生静态反射的便捷程度相去甚远。

举个例子，获取每个结构体的信息，代码如下：

```
1 #include <iostream>
2 #include <string>
3
4 struct person {
5     int age;
6     std::string name;
7 };
```

¹<https://www.circle-lang.org>

```

8
9
10 template<typename T>
11 void print_members() {
12     std::cout << @type_string(T) << ":\n";
13     @meta for(int i = 0; i < @member_count(T); ++i) {
14         std::cout << i << " " << @member_type_string(T, i)
15             << " " << @member_name(T, i) << "\n";
16     }
17 }
18
19 int main() {
20     print_members<person>();
21 }
```

前缀为@的就是扩展的反射工具集，其中@type_string 用于得到类型的字符形式，@member_count 用于得到类型的成员个数，@member_type_string 用于得到成员类型的字符形式，@member_name 用于得到成员的名称。

Circle 是静态反射，故所有操作都需要在编译期完成，而普通for发生于运行期。在其前面添加一个@meta 关键字，就能使for发生于编译期。

编译输出，将会得到：

```
$ circle refl.cxx && ./refl
person:
0 int age
1 std::string name
```

Circle 还提供了另外一种更加简便的方式，上述代码还可以这样写：

```
template<typename T>
void print_members() {
    std::cout << @type_string(T) << ":\n";
    (std::cout << int... << " " << @member_type_strings(T)
        << @member_names(T) << "\n")...
}
```

所有的反射成员工具集都变成了复数形式，用法很像 *fold expressions*。其中的int... 称为 *pack index*，这是在简化标准中的std::index_sequence。类有权限之分，Circle 也支持按照不同的权限来获取类成员，例如：

```

1 struct pointer {
2     public: float x;
3     protected: float y;
4     private: float z;
5 };
6
```

```

7
8 int main() {
9     //print_members<person>();
10    //print_enumerations<Days>();
11    (std::cout << int... << ":" << @member_decl_strings(pointer, 1) << "\n---\n")...
12    (std::cout << int... << ":" << @member_decl_strings(pointer, 2) << "\n---\n")...
13    (std::cout << int... << ":" << @member_decl_strings(pointer, 4) << "\n")...
14 }

```

其中1表示只打印public成员，2表示只打印protected成员，4表示只打印private成员，所以输出为：

```

0: float x
---
0: float y
---
0: float z

```

完整的权限数字是0-7，其中0表示什么都没有，3表示所有public 和protected成员，5表示所有public和private成员，6表示所有protected和private成员，7表示所有成员。

第一章中，曾展示过标准静态反射转换枚举到字符串的实现，Circle 亦可轻松实现。

```

1 enum class Days {
2     Monday = 1,
3     Tuesday,
4     Wednesday,
5     Thursday,
6     Friday,
7     Saturday,
8     Sunday
9 };
10
11 template<typename Enum>
12 requires std::is_enum_v<Enum>
13 std::string enum_to_string(Enum value) {
14     switch(value) {
15         @meta for enum(Enum e : Enum) {
16             case e:
17                 return @enum_name(e);
18         }
19
20         default:
21             return "<unnamed>";
22     }
23 }

```

```

24
25 template<typename Enum>
26 void print_enumerations() {
27     std::cout << @type_string(Enum) << ":\n";
28     (std::cout << int... << ":" << enum_to_string(@enum_values(Enum)) << '\n')...
29 }
30
31 int main() {
32     //print_members<person>();
33     print_enumerations<Days>();
34 }
```

运行将会输出所有枚举成员：

```
$ circle refl.cxx && ./refl
Days:
0: Monday
1: Tuesday
2: Wednesday
3: Thursday
4: Friday
5: Saturday
6: Sunday
```

其中遍历Enum 主要用的是@meta for enum，@enum_name 用于获取枚举的字符值。将其写在switch当中，则会在编译时自动为每个枚举成员展开相应的case语句。

最简单的方式是直接使用@enum_name 输出，此时就无需编写enum_to_string 函数：

```
template<typename Enum>
void print_enumerations() {
    std::cout << @type_string(Enum) << ":\n";
    (std::cout << int... << ":" << @enum_name(@enum_values(Enum)) << '\n')...
}
```

只需要一行代码，是不是超级简洁呢！相反，亦可从字符串来产生枚举类型，如下代码：

```

1 template<typename Enum>
2 std::optional<Enum> string_to_enum(const char* s) {
3     @meta for enum(Enum e : Enum) {
4         if(0 == strcmp(@enum_name(e), s))
5             return e;
6     }
7     return { };
8 }
9
10 int main() {
```

```

11     //print_members<person>();
12     //print_enumerations<Days>();
13     auto value = string_to_enum<Days>("Sunday");
14     assert(*value == Days::Sunday);
15 }
```

可以看到，灵活性非常强。

32.2 自定义 Attributes

反射和自定义 Attributes 组合起来，程序能够产生极大的灵活性。

(注：不了解 Attributes 可参考：那些值得使用的标准 Attributes)

若大家使用过 Java 的注解，就一定能够明白这东西的意义，Java 的许多开源框架，如著名的 Spring，底层便基于注解与反射。

Circle 中自定义 Attributes 并不复杂，一个简单的例子：

```

1 // declare user attributes
2 using color [[attribute]] = const char*;
3 using country [[attribute]] = const char*;
4
5 struct book {
6     [[.color="red"]]] std::string title;
7     [[.country="China"]]] std::string author;
8     int page_count;
9 };
10
11
12
13 int main() {
14     // print_members<person>();
15     // print_enumerations<Days>();
16     // (std::cout << int... << ":" << @member_decl_strings(pointer, 1) << "\n---\n")...
17     // (std::cout << int... << ":" << @member_decl_strings(pointer, 2) << "\n---\n")...
18     // (std::cout << int... << ":" << @member_decl_strings(pointer, 4) << "\n")...
19     book b;
20     std::cout << @attribute(b.@member_value(0), color);    // output: red
21     std::cout << @attribute(b.@member_value(1), country); // output: China
22
23 }
```

自定义 attribute 有两种方式，语法如下：

```

using attrib-name [[attribute]] = attrib-type;
using attrib-name [[attribute]] = typename;
```

第一种是非类型 *attribute*, 可以指定值; 第二种是类型 *attribute*, 可以指定类型。

例子中属于第一种方式, 分别为 `color` 和 `country` 进行赋值, 值类型为 `const char*`。赋值之后, 通过 `@attribute` 便能够访问所定义的 *attribute* 值。

枚举也支持自定义 *attributes*, 可以改写前面的代码如下:

```

1  using vacation [[attribute]] = const char*;
2
3  enum class Days {
4      Monday = 1,
5      Tuesday,
6      Wednesday,
7      Thursday,
8      Friday,
9      Saturday [[.vacation="true"]],
10     Sunday [[.vacation="true"]]
11 };
12
13 template<typename Enum>
14 requires std::is_enum_v<Enum>
15 std::string enum_to_string(Enum value) {
16     switch(value) {
17         @meta for enum(Enum e : Enum) {
18             case e:
19                 if constexpr (@enum_has_attribute(e, vacation))
20                     return @enum_attribute(e, vacation);
21                 else
22                     return @enum_name(e);
23         }
24
25         default:
26             return "<unnamed>";
27     }
28 }
29
30 int main() {
31     // print_members<person>();
32     // print_enumerations<Days>();
33     // (std::cout << int... << ":" << @member_decl_strings(pointer, 1) << "\n---\n")...
34     // (std::cout << int... << ":" << @member_decl_strings(pointer, 2) << "\n---\n")...
35     // (std::cout << int... << ":" << @member_decl_strings(pointer, 4) << "\n")...
36     // book b;
37     // std::cout << @attribute(b.@member_value(0), color);    // output: red
38     // std::cout << @attribute(b.@member_value(1), country); // output: China

```

```

39     (std::cout << enum_to_string(@enum_values(Days)) << "\n---\n")...;
40
41 }

```

输出将为：

```

1 Monday
2 ---
3 Tuesday
4 ---
5 Wednesday
6 ---
7 Thursday
8 ---
9 Friday
10 ---
11 true
12 ---
13 true
14 ---

```

有了这种自定义 *Attributes* 支持，就可以根据 *attributes* 的值来动态处理相应的成员。

比如对于一个服务器，为每种类型的消息函数添加 *attributes*，便可以根据该 *attribute* 值自动实现逻辑分派。

32.3 Typed enums

读过《C++ 设计新思维》的应该知道，其中包含一个强大的工具 *TypeList*，Circle 中内建了这个类型，名为 *typed enums*。

普通的 `enum` 是整型值，而 *typed enums* 中存放的是类型，不像 *Loki* 中的 *TypeList*，此内建类型并非采用链表来连接每个类型，而是采用数组。

来看看其基本用法，代码如下：

```

1 enum typename type_list {
2     int,
3     double,
4     char*,
5     char,
6     std::string
7 };
8
9 template<typename type_t>
10 void print_typed_enum() {
11     std::cout << @type_string(type_t) << '\n';

```

```

12     (std::cout << int... << ":" << @enum_type_strings(type_t) << '\n')...;
13 }
```

通过`enum typename`来定义一个typed enums，访问起来与普通`enum`一样。调用输出将为：

```

type_list
0: int
1: double
2: char*
3: char
4: std::string
```

我们还可以通过反射来增加类型，扩展`type_list`，如下：

```

1 enum typename type_list_join {
2     float;
3     int[5];
4
5     // 使用包展开定义类型
6     @enum_types(type_list)...;
7
8     // 逆序展开
9     @enum_types(type_list)...[::-1] ...;
10};
```

这个新的typed enums 包含着两个自己的类型，其余类型通过包展开添加`type_list`中的类型。其中`...[::-1]`用于指定包展开的顺序为逆序展开，也可以指定特定区间的类型，完整语法是`...[start:stop:step]`，分别表示开始位置、结束位置、步长，非常灵活。

故上述代码的输出将为：

```

1 type_list_join
2 0: float
3 1: int[5]
4 2: int
5 3: double
6 4: char*
7 5: char
8 6: std::string
9 7: std::string
10 8: char
11 9: char*
12 10: double
13 11: int
```

除此之外，还可以对typed enums 应用算法，比如对`type_list`中的所有类型进行排序，

```

1 enum typename sorted_type_list {
2     @meta std::array types {
```

```

3     std::make_pair<std::string, int>(
4         @enum_type_strings(type_list_join),
5         int...
6     )...
7 };
8
9     @meta std::sort(types.begin(), types.end());
10
11    @enum_type(type_list_join, @pack_nontype(types).second) ...;
12 }

```

此处，先定义了一个`std::array`类型的`types`，元素类型为一个`pair`元素，第一个键为`enum`的名称，第二个键为名称所对应的索引。

之后，对`types`进行排序，根据排序结果，就能够得到排好序的索引。借助`@pack_nontype`可以将容器中的值表示为非类型参数包，因此可以进行展开索引。

因此输出将为：

```

1 sorted_type_list
2 0: char
3 1: char
4 2: char*
5 3: char*
6 4: double
7 5: double
8 6: float
9 7: int
10 8: int
11 9: int[5]
12 10: std::string
13 11: std::string

```

Loki 中为`TypeList`编写的诸多算法，如索引式访问、唯一判断、存在查询等等，在`Circle`中基于反射这些都可以轻松实现。

若要遍历类型，除了前面的`for-enum`，`Circle`还提供了`for-typename`。可以遍历类型列表，亦可遍历`typed enums`，用法如下：

```

1 template<typename... types>
2 void print_types() {
3     @meta for typename(type_t : { types... }) {
4         std::cout << @type_string(type_t) << '\n';
5     }
6 }
7
8 template<typename type_list>
9 void print_typed_enum() {

```

```

10     // std::cout << @type_string(type_t) << '\n';
11     // (std::cout << int... << ":" << @enum_type_strings(type_t) << '\n')...
12     @meta for typename(type_t : enum type_list)
13         std::cout << @type_string(type_t) << '\n';
14     }
15
16 print_types<int, char, float, double, std::string>();
17 print_typed_enum<sorted_type_list>();

```

总而言之，typed enums 配上反射非常好用，操纵类型和容器一样方便。

32.4 动态命名

若想动态定义变量名，当前只能根据宏来生成，但局限很大，能力甚微。而有了反射，动态命名应该轻而易举，Circle 便支持此特性，例如：

```

1 template<typename... type_t>
2 struct tuple_t {
3     const char* @("var");
4     type_t @(int...) ...;
5 };
6
7 // 遍历所有定义成员全称
8 @meta puts(@member_decl_strings(tuple_t<int, char, double>))...

```

通过@() 操作符可以连接字符串或数字来动态产生变量名，而指定 *pack index* 则会动态产生_0, _1... 这样的名称。

因此，上述代码其实是为tuple_t 动态定义了如下成员，

```

const char* var;
int _0;
char _1;
double _2;

```

这个特性，将为自动产生代码带来极大的活力。

32.5 动态生成类

基于 Circle 的这种强大反射能力，可以真正实现「自动生成类」。比如将类型信息保存成 JSON 数据，那么根据这些数据，依赖反射就可以动态生成类型。让我们先准备 JSON 格式的类型数据，代码如下：

```

@meta json j;
@meta j["types"]["name"] = "person";
@meta j["types"]["members"] = { {{"name", "age"}, {"type", "int"}},
                                {"name", "name"}, {"type", "std::string"} };

```

这里借助了 *nlohmannjson* 库，由于是静态反射，因此数据也必须在编译期产生，所以前面都加上了 @meta。以上定义的数据为，

```

1  {
2      "types": {
3          "members": [
4              {
5                  "name": "age",
6                  "type": "int"
7              },
8              {
9                  "name": "name",
10                 "type": "std::string"
11             }
12         ],
13         "name": "person"
14     }
15 }
```

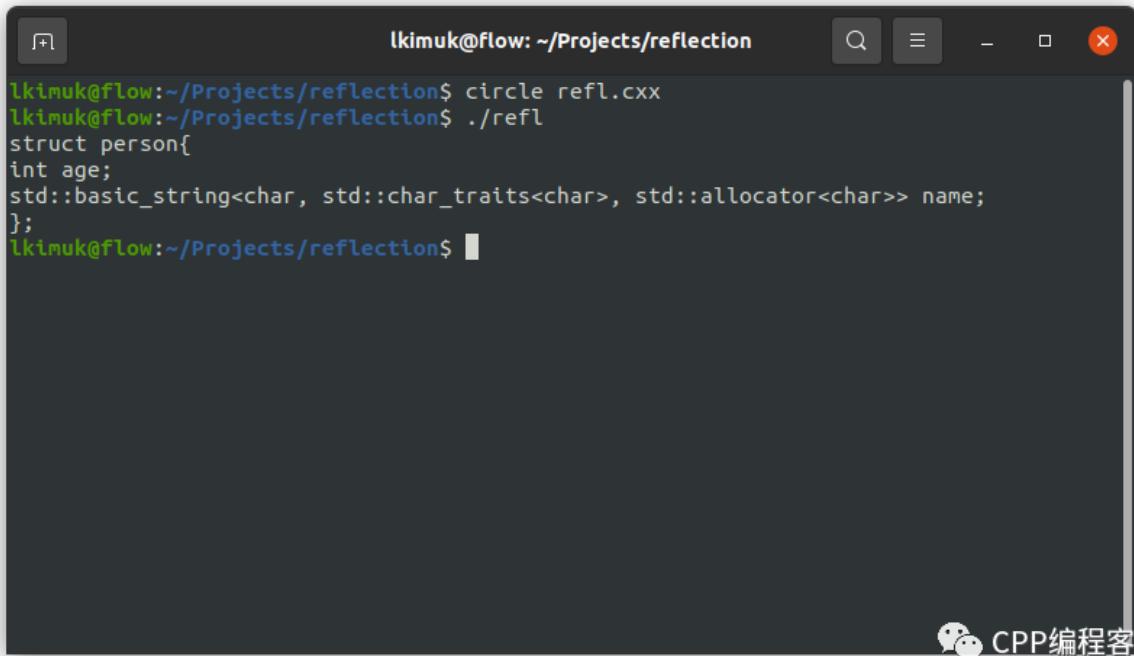
就是前面的 `person` 类，不过如今是通过 JSON 和反射来动态定义该类。然后，通过反射来动态生成类，代码如下：

```

1 @meta json& types = j["types"];
2 @meta struct @types["name"] {
3     @meta for(json& members : types["members"]) {
4         @type_id(members["type"]) @members["name"];
5     }
6 };
7
8 enum typename type_list {
9     @type_id(types["name"]);
10 };
11
12
13 int main() {
14
15     @meta for typename(type_t : enum type_list) {
16         std::cout << "struct " << @type_string(type_t) << "{\n";
17         (std::cout << @member_decl_strings(type_t) << ";\n")...
18         std::cout << "};\n";
19     }
20 }
```

通过动态命名操作符 @() 可以根据字符串产生类名，再通过 @type_id 定义类型，便可完整地定义

类型的所有成员。为了访问类型信息，将其置入`typed enums`，接着就能够使用`for-typeinfo`进行打印类型信息。因此输出如图32.1。



The screenshot shows a terminal window titled "lkimuk@flow: ~/Projects/reflection". The user has run the command "circle refl.cxx" followed by "./refl". The output displays the definition of a struct named "person" with a member "age" and a member variable "name" of type "std::basic_string<char, std::char_traits<char>, std::allocator<char>>". The "name" member is highlighted in blue, indicating it is being processed or reflected upon.



图 32.1: 动态生成类

这种层次的反射能力非常强大，像 ORM、远程调用等等需求实现起来轻而易举。

32.6 总结

Circle 的反射扩展，是目前除标准之外能力最强的。正规项目中肯定无法使用这些扩展，但是可以一睹反射所应该具备的能力。

事实上，C++ 反射一直在铺路，像是依次增加的编译期关键字，就是在为静态反射做准备。标准中将加入的反射，只会比 Circle 扩展的反射能力更强，其中的有些概念 Circle 当中也没有，支持的功能更多，只是还须不少时间。

只能说，标准现在所支持的反射预备特性太少了，即便是 reflection ts，使用起来跟 Circle 的体验都相差甚远。故想用上标准反射，C++26 可能都是早的了：P

Chapter 33

C++ 反射 第四章 标准

• 里缪 2022-05-08



到了此章，才算是正式进入「C++ 反射」的主题。

其实从第一章就已然确定了该系列的结构。鉴于「静态反射」还未进入标准，于是有了第二章 T0 层反射，这是当前项目中可以使用的；再有了第三章 T1 层反射，这是当前相对完整的反射实践，标准亦会参考 Circle。由于缺少底层机制，因此 T0 层反射几乎属于玩具级别，而 Circle 只供语法参考，无法在正式项目中使用。

这篇我们正式进入 C++ 标准反射。标准反射最早可于 C++26/29 进入标准，故本章几乎全是比較新的概念。

33.1 C++ 静态反射与元编程的关系

静态反射加入标准，将会使 C++ 元编程进入一个全新的阶段。

为什么这样说呢？

在 C++ 中，谈论元编程，一般我们是指编译期的编程。其发展可以分为三个阶段。

第一个阶段，属于模板时期，起于 C++98。模板可以作为编译期计算的工具，对 C++ 影响深远，借其可支持泛型编程，优化代码。缺点是需要对语言具有深入的理解，才能够使用模板来设计与维护代码。

第二个阶段，属于 *Constant Expressions* (常量表达式) 时期，起于 C++11。这使得编写编译期代码更加便捷，C++11 开始，通过使用 `constexpr` 关键字，便可以令某些运算发生于编译期，C++20 又添加了 `consteval` 与 `constinit`，对编译期计算提供了更多支持。基本用法见：Differences between keywords `constexpr`, `consteval` and `constinit`

第三个阶段，则属于静态反射时期，SG7 仍在努力中。大概从 2010 年，静态反射的研究工作便已启动，最终产生了一个 TS 版本 (N4766)，此时属于 *type-based* 反射。后来，出于种种原因，SG7 转而支持之前就已提出的 *value-based* 反射 (P0425r0)。这个阶段将影响深远，极大增强 C++ 元编程的能力，改变大家编程的方式。

本篇文章，就是带领大家进入第三阶段，学习 *value-based* 反射的最新成果。需要提醒大家的是，静态反射本身强调的是反射能力，只有这种能力，根本不够。因此，伴随静态反射还增加了一些其他提案，比如 *Expansion Statements* 用于遍历复数式反射元信息，源码注入用于支持更加强大的产生式元编程。

静态反射加上这些相关提案，才真正构成了反射大家族，这才是第三阶段的 C++ 元编程。

33.2 实践环境的选择

只是纸上谈兵，社区自然没甚激情，所以 SG7 提供了一些基于 *reflection ts*（后期 *value-based* 版本）的实现，以在社区激起一些浪花。

clang 提供有一个支持 *reflection ts* 的版本，Compiler Explorer 上可以直接使用。

然而这个版本不足以支持本篇内容，因为我们还需要 *Expansion Statements* 以及源码注入这些提案的支持，因此选择基于 llvm 的另一个版本：*lock3*。

lock3 版本现在也没有维护了，但仍是当前最完善的实现版本，编译得半天，大家直接使用 Compiler Explorer 在线版本就可以，复制链接<https://godbolt.org/z/nc34GKvMMt>直接进入。

链接当中我已经提前写好了反射所需的全部头文件，这些头文件名称当然也不标准，一个实现一个样。

```

1 #include <experimental/meta>
2 #include <experimental/compiler>
3
4 using namespace std::experimental;
5
6 int main() {
7
8 }
```

反射属于元编程的第三个阶段，相关特性最终都会包含在<meta> 头文件当中。关于该头文件的具体内容，后面有一节专门介绍。

完成了前置知识，下面开始正式进入标准反射的内容。

注意！本文讲解的内容基于最新的语法，而编译器的实现仍是较旧或有不符合提案的语法，因此同一个概念，讲解的语法与实际编写代码的语法，存在不一样的形式。各位要记住的是最新的语法，旧语法只是编写代码时不得不向编译器做出的妥协。

33.3 The \wedge operator and Splicing

这两个概念在第一章通识中已经介绍过，对应 *reflection* 与 *reification*。

reflection 表示从类型得到「类型元信息」这个动作，是从具体到抽象、自下而上的结构；而 *reification* 表示从「类型元信息」再次得到类型这个动作，是从抽象到具体、自上而下的结构。

这两个是反射通用的概念，在 C++ 反射中 *reflection* 对应于“ \wedge operator”，读作 *lifting operator*，表示向上获取类型元对象；*reification* 对应于 *splicing*，语法为”[: refl :]”，称为 *splice construct*，表示重新具体化对象。

也就是说，“ \wedge operator”是进入反射世界的钥匙，而“[: refl :]”则是回到现实世界的钥匙。

可以再次回到第一章的示例中熟悉一下这些概念。

```

1 #include <meta>
2 template<Enum T>
3 std::string to_string(T value) {
4     template for (constexpr auto e : std::meta::members_of(^T)) {
```

```

5     if([:e:] == value) {
6         return std::string(std::meta::name_of(e));
7     }
8 }
9 return "<unnamed>";
10 }

```

细节第一章讲了，不再赘述。这个例子本身没什么用，主要是为了串联起诸多概念。不过，这个例子当前编译不过，因为这两个概念的新语法当前没有任何编译器支持。

那么这个例子在 lock3 下如何编写呢？代码如下：

```

1 template<typename T>
2 requires std::is_enum_v<T>
3 consteval const char* enum_to_string(T value) {
4     template for (constexpr auto e : std::meta::members_of(reflexpr(T))) {
5         if(idexpr(e) == value) {
6             return __concatenate(std::meta::name_of(e));
7         }
8     }
9     return __concatenate("<unnamed>");
10 }
11
12 enum class Colors : unsigned char {
13     Red,
14     Green,
15     Blue,
16     Yellow,
17     Black
18 };
19
20 int main() {
21     constexpr const char* color_name = enum_to_string(Colors::Black);
22     constexpr auto __dummy = __reflect_print(color_name);
23 }

```

为什么反射学起来很乱呢？就是因为概念之间变化太快了，一会叫这个，一会叫那个。

lock3 当前实现的 *lifting operator* 还是最早使用的占位符：`reflexpr()`。而 *splicing* 的支持都是自己提供的，不像“[: ... :]”语法具有一致性，它提供了好几个操作符来支持不同的情况，`idexpr()` 就是其中之一。

所有的操作都发生于编译期，而 C++ 还不支持`constexpr string`（虽然可以借助`std::string_view`），但是 lock3 提供的`__concatenate()` 用来产生编译期字符串要更加好用。此外，要在编译期输出`constexpr string`，lock3 提供了`__reflect_print()`。

反射结果称为「元对象」(*metaobject*)，也就是反射类型，定义为：

```
namespace std::meta {
```

```
using info = decltype(^void);
}
```

这是一个唯一的编译期常量值，所有的反射相关函数，参数都有`meta::info`。

通过 *lifting operator* 就能得到反射类型，再通过`meta::members_of()` 来根据元对象获取类型的所有成员，它的返回值不止一个，若要遍历就得使用 *expansion statements*。

运行上述程序，最终将会在编译期输出枚举值的字符形式，如图33.1。

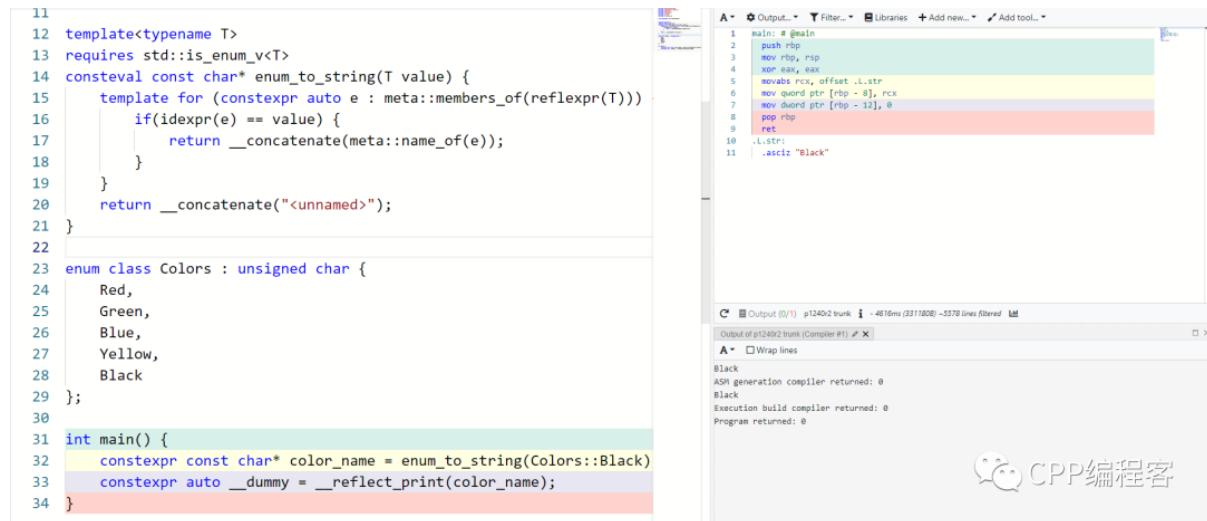


图 33.1: Lock3 enum to string

33.4 标准元编程库

标准元编程库，头文件为`<meta>`，其中的所有函数称为「元函数」(*metafunctions*)。

元编程库主要包含两部分内容，第一部分包含元编程的一些通用组件，第二部分包含反射 TS 中的元函数。

首先来看通用组件部分。通用组件提供许多针对类型的工具，这些工具大多来自`<type_traits>`，只是将那些特性加到了 *value-based* 反射中来，以支持元对象（即反射类型`meta::info`）作为参数。

比如`remove_const`，定义为：

```
consteval info remove_const(info type) {
    detail::require_type(type);
    return __reflect(detail::query_remove_const, type);
}
```

那么如何使用呢？其实和`type_traits`中一样，只是针对的是元对象，举个例子：

```
constexpr auto MyInt = meta::remove_const(reflexpr(const int));
typename(MyInt) var = 10;
var = 42;
```

这里通过 *lifting operator* 得到`const int`的元对象，再通过元函数移除该类型的`const`修饰，便得到了一个`MyInt`元对象。通过`typename()`可以将元对象 *splicing*为一个类型，这代码就相当于`int var = 10`。

补充一下，上述 *splicing* 语法最新写法为`typename[:MyInt:]`。

因为`<type_traits>` 大家基本都用过，只是将那些工具引入到反射里面，所以就不浪费篇幅介绍，以后的实际应用中再作解释。

接着来说第二部分，这些工具皆来自反射 TS，都是新的元函数。比如下面的例子，检测类成员权限：

```

1 struct player {
2     public:
3         float position_x;
4     private:
5         float position_y;
6     protected:
7         float position_z;
8 };
9
10 int main() {
11     constexpr bool pub = meta::is_public(reflexpr(player::position_x));
12     constexpr bool pri = meta::is_private(reflexpr(player::position_x));
13     constexpr bool pro = meta::is_protected(reflexpr(player::position_x));
14 }
```

在反射之前，C++ 元编程并不具备这样的能力。上面这种返回`bool`值的元函数，属于 *predicates* 一类，这个类别里面还有许多，列举一些如下：

```

1 std::meta::is_unnamed
2 std::meta::is_scoped_enum
3 std::meta::is_declarad_constexpr
4 std::meta::is_consteval
5 std::meta::is_static
6 std::meta::is_inline
7 std::meta::is_deleted
8 std::meta::is_defaulted
9 std::meta::is_explicit
10 std::meta::is_override
11 std::meta::is_pure_virtual
12 std::meta::is_class_member
13 std::meta::is_local
14 std::meta::is_namespace
15 std::meta::is_template
16 std::meta::is_type
17 std::meta::is_incompltete_type
18 std::meta::is_closure_type
19 std::meta::has_captures
20 std::meta::has_default_ref_capture
21 std::meta::has_default_copy_capture
```

```
22 std::meta::is_simple_capture
23 std::meta::is_ref_capture
24 std::meta::is_copy_capture
25 std::meta::is_explicit_capture
26 std::meta::is_init_capture
27 std::meta::is_function_parameter
28 std::meta::is_template_parameter
29 std::meta::is_class_template
30 std::meta::is_alias
31 std::meta::is_alias_template
32 std::meta::is_enumerator
33 std::meta::is_variable
34 std::meta::is_variable_template
35 std::meta::is_static_data_member
36 std::meta::is_nonstatic_data_member
37 std::meta::is_bit_field
38 std::meta::is_base_class
39 std::meta::is_direct_base_class
40 std::meta::is_virtual_base_class
41 std::meta::is_function
42 std::meta::is_function_template
43 std::meta::is_member_function
44 std::meta::is_member_function_template
45 std::meta::is_static_member_function
46 std::meta::is_static_member_function_template
47 std::meta::is_nonstatic_member_function
48 std::meta::is_nonstatic_member_function_template
49 std::meta::is_constructor
50 std::meta::is_constructor_template
51 std::meta::is_destructor
52 std::meta::is_destructor_template
53 std::meta::is_lvalue
54 std::meta::is_xvalue
55 std::meta::is_prvalue
56 std::meta::is_glvalue
57 std::meta::is_rvalue
58 std::meta::has_ellipsis
59 std::meta::is_member_function_type
60 std::meta::has_default
```

下面再来看元函数的另一个类别，单数形式。意思很明显，此类元函数只会返回一个结果，比如前面使用过的`meta::name_of()`用来获取类型的名称。这里再给个小例子，如何打印类型名称：

```
1 int a = 10;
```

```

2 constexpr auto r = meta::type_of(refexpr(a));
3 constexpr auto n = meta::name_of(r);
4 std::cout << n; // output: int

```

这个类别也有一些元函数，不过 *lock3* 中实现的不多，列举一些如下：

```

1 std::meta::source_location_of
2 std::meta::name_of
3 std::meta::display_name_of
4 std::meta::entity
5 std::meta::type_of
6 std::meta::parent_of
7 std::meta::current_function
8 std::meta::current_class_type
9 std::meta::byte_offset_of
10 std::meta::bit_offset_of
11 std::meta::byte_size_of
12 std::meta::bit_size_of
13 std::meta::this_ref_type

```

这里列举的很多元函数编译器目前不支持，大家注意一下。

最后来看元函数的另一个类别，复数形式。顾名思义，就是返回多个结果的元函数。这里举个打印类成员的例子：

```

1 struct player {
2     static int x;
3     double y;
4 private:
5     float z;
6 protected:
7     void foo() {}
8 };
9
10 template<typename T>
11 void print_members() {
12     constexpr auto members = meta::members_of(refexpr(T));
13     template for (constexpr auto m : members) {
14         constexpr auto __dummy = __reflect_pretty_print(m);
15     }
16 }
17
18 int main() {
19     print_members<player>();
20 }

```

在编译期将会输出：

```
static int x
double y
float z
void foo() {
}
```

打印所有成员，包括`private`与`protected`成员。

这里`meta::members_of()`就是一个返回复数式反射信息的元函数，遍历这种元函数的结果，就需要使用 *expansion statements*。`__reflect_pretty_print()`是 *lock3* 提供的另一个编译期输出工具，参数为元对象，可以直接打印类型的实际声明形式。`meta::members_of()`还有第二个参数，可以指定 *predicates* 一类的元函数，作为约束条件。比如：

```
// 获取所有非静态数据成员
meta::members_of(reflexpr(T), meta::is_nonstatic_data_member);
// 获取所有私有成员
meta::members_of(reflexpr(T), meta::is_private);
// 获取所有保护成员
meta::members_of(reflexpr(T), meta::is_protected);
```

同样，我们也可以获取函数的参数，代码如下：

```
1 void foo(int a, double b, const std::string& c) {
2 }
3
4 void print_parameters() {
5     constexpr auto parameters = meta::parameters_of(reflexpr(foo));
6     template for (constexpr auto p : parameters) {
7         constexpr auto __dummy = __reflect_pretty_print(p);
8     }
9 }
10
11 int main() {
12     print_parameters();
13 }
```

输出将为：

```
int a
double b
const std::string c
```

以上大概就是对`<meta>`库的一个整体介绍，下面将进入对于元编程来说，更加重要的一个模块：源码注入。

33.5 源码注入

什么是源码注入？为什么它如此重要呢？

我们使用反射，最主要的是使用「产生式元编程」，反射属于基本组件，让我们能够操纵「类型元信息」。而在此之上，需要一些其他特性，来支持使用反射进行产生式元编程。源码注入，就是伴随反射而来的提案，使得我们可以编写产生代码的代码。

还是以一个简单的例子开始：

```

1 struct X {
2     consteval {
3         for (int num = 0; num < 10; ++num)
4             -> fragment struct {
5                 int unqualid("variable_", %{num});
6             };
7     }
8 }
```

此处便使用源码注入，自动生成了以下代码：

```

1 struct X {
2     int variable_0;
3     int variable_1;
4     int variable_2;
5     int variable_3;
6     int variable_4;
7     int variable_5;
8     int variable_6;
9     int variable_7;
10    int variable_8;
11    int variable_9;
12 }
```

我们要进行注入的代码区域，称为 *metaprogram*，语法如下：

```
consteval {
    ...
};
```

可以将 *metaprogram* 当作是一个不需要参数的 *consteval* 函数，编译时会自动执行。

metaprogram 中有一些其他操作，比如例子中的 *for* 自动产生 *num*，这些操作是不需要注入的，真正需要注入的语句，只要通过「注入语句」的语法“->”指定，就可以将这些代码注入到源码中去。

注入语句跟着的需要注入的代码片段称为 *fragments*，是一个表达式，类型为元对象 (*meta::info*)。

fragments 有许多种类，比如 *namespace fragments*, *class fragments*, *enumeration fragments*, *block fragments*，分别表示注入到不同的源码中去。例子中使用的 *fragment struct*，就属于 *class fragments*，也存在 *fragment class*，区别与 *struct* 和 *class* 的区别一样。

注意，*class fragments* 并不是描述一个真正的类，它仅仅是描述类中的成员。也就是说，不用给这个类起名字，起了也会被替换掉，最终被注入的只是这个类的 *body*。*class fragments* 只能注入到类里面。

那么如何在 *fragments* 中引入外部的变量呢？

这就需要使用 *unquote operator* 了，语法为 "%{ ... }"，它允许引用局部变量作为 *fragments* 中的变量名或类型名。

而 *unqualid operator*，可以让我们从字符串组合新的代码。

前面说过，*fragments* 的类型为元对象，那么其实是可以分开定义的，这是再举个 *fragment enumeration* 的例子：

```

1 constexpr meta::info rgb = fragment enum {
2     red, blue, green
3 };
4
5 enum class color {
6     consteval -> rgb
7 };

```

这里将会把 *fragments* 中的值注入到枚举的源码中去，*metaprogram* 只有一行的情况下，就可以省略 "{}"。

因此上面源码注入之后的代码为：

```

1 enum class color {
2     red,
3     blue,
4     green
5 };

```

配合反射，可以实现更多强大的源码注入能力，因为反射的类型也是元对象，所以可以直接注入。看个简单的例子：

```

1 struct A {
2     void foo() {}
3 };
4
5 struct X {
6     consteval -> reflexpr(A::foo);
7 };

```

这样就轻松地将 A 的成员函数，注入到了 X 之中。同时，也可以在注入的过程中，修改原始定义，比如：

```

1 struct X {
2     consteval {
3         meta::info foo_refl = reflexpr(A::foo);
4         meta::make_constexpr(foo_refl);
5
6         const char* name = meta::name_of(foo_refl);
7         meta::set_new_name(foo_refl, __concatenate("constexpr_", name));
8     }

```

```

9         -> foo_refl;
10    }
11 };

```

这就相当于如下声明：

```

struct X {
    constexpr void constexpr_foo() {}
};

```

可以看到，反射配合源码注入，具备强大的产生式元编程能力。关于源码注入，编译器支持的也不算全，本篇暂时只介绍这么多内容，更多内容以后再写。

以下各节，展示一些不太复杂的使用情境。

33.6 自动生成 getters 和 setters

通过标准反射与源码注入，可以轻松为类成员实现getters 和setters 函数。当前有如下类：

```

1 struct book {
2     std::string title;
3     std::string author;
4     int page_count;
5
6     consteval {
7         gen_members(reflexpr(book));
8     }
9 };

```

通过使用*fragments*，我们想自动生成如下代码：

```

1 struct book {
2     std::string title;
3     std::string author;
4     int page_count;
5
6     std::string get_title() const {
7         return title;
8     }
9
10    void set_title(const std::string& title) {
11        this->title = title;
12    }
13
14    std::string get_author() const {
15        return author;
16    }

```

```

17
18     void set_author(const std::string& author) {
19         this->author = author;
20     }
21
22     int get_page_count() const {
23         return page_count;
24     }
25
26     void set_page_count(const int& page_count) {
27         this->page_count = page_count;
28     }
29 };

```

那么首先，定义 `gen_members()` 函数，代码如下：

```

1 consteval void gen_members(meta::info cls) {
2     auto members = meta::members_of(cls, meta::is_nonstatic_data_member);
3     for (meta::info member : members) {
4         gen_member(member);
5     }
6 }

```

通过元函数遍历出类的所有非静态数据成员，再为每个数据成员生成 `getter` 和 `setter` 函数。`gen_member()` 函数的定义如下：

```

1 consteval void gen_member(meta::info m) {
2     -> fragment struct {
3         typename(meta::type_of(%{m}))
4         unqualid("get_", meta::name_of(%{m}))() const {
5             return unqualid(meta::name_of(%{m}));
6         }
7     };
8
9     -> fragment struct {
10        void unqualid("set_", meta::name_of(%{m}))((
11            const typename(meta::type_of(%{m}))& unqualid(meta::name_of(%{m}))) {
12            this->unqualid(meta::name_of(%{m})) = unqualid(meta::name_of(%{m}));
13        }
14    };
15 }

```

通过定义两个 *fragments*，利用元编程库中的组件，与源码注入的相关特性，就能够达到自动产生代码的能力。这个 *fragments* 具备可复用性，对所有的类都适用。

这种对当前类生成一些新代码，有一种更推荐的方式，称为 *Metaclasses*，这个概念属于源码注入。上述例子以这种方式编写，代码如下：

```

1 consteval void gen_members(meta::info cls) {
2     auto members = meta::members_of(cls, meta::is_nonstatic_data_member);
3     for (meta::info member : members) {
4         ->member;
5
6         gen_member(member);
7     }
8 }
9
10 struct(gen_members) book {
11     std::string title;
12     std::string author;
13     int page_count;
14 };

```

Metaclasses 就是一个 *metaprogram*, 不过它强调的是从一个类的原型 (*prototype*) 来为该类产生新的代码。

Metaclasses 这种方式的语法为”`struct(...)`”, 这样就无需在类内编写 *metaprogram*, 这种方式要更加安全, 编写的代码量也更少。

这里有一点需要注意, 编译器在实例化该定义时, 会有一个隐藏的 `self` 元对象, 还有一个处于匿名空间之中的 `book` 类, 这个就是原型类。所有声明的成员处于原型类中, `self` 当中没有, 因此要使用”`->member`”将这些成员重新产生出来, 否则 `book` 类当中找不到声明的那些成员。

此外, 其实这个功能的实现, 需要另一个特性配合才更好用, 那就是「自定义 Attributes」, 存在相关提案, 但是目前没法使用。

33.7 自动生成 SQL 语句

这个功能是第二章中出现的例子, 现在更新为标准 C++ 反射版本。代码如下:

```

1 consteval const char* to_sql(meta::info type) {
2     if(type == reflexpr(int))
3         return "INTEGER";
4     else if(type == reflexpr(std::string))
5         return "TEXT";
6
7     return "UNKNOWN_TYPE";
8 }
9
10 consteval const char* create_column(meta::info member) {
11     return __concatenate(meta::name_of(member),
12         " ", to_sql(meta::type_of(member)));
13 }
14
15 template<meta::info Class>

```

```

16 consteval const char* create_table() {
17     const char* table_name = meta::name_of(Class);
18
19     const char* sql = __concatenate("CREATE TABLE ", table_name, "(");
20     bool first_seen = false;
21     for(meta::info member : meta::data_member_range(Class)) {
22         if(first_seen)
23             sql = __concatenate(sql, ", ");
24
25         sql = __concatenate(sql, create_column(member));
26         first_seen = true;
27     }
28     return __concatenate(sql, ")");
29 }
```

代码逻辑第二章讲过，这里不再赘述。测试代码，将会输出：

```

1 struct person {
2     int age;
3     std::string name;
4 };
5
6 int main() {
7     std::cout << create_table<refexpr(person)>();
8 }
9
10 // 输出: CREATE TABLE person(age INTEGER, name TEXT);
```

33.8 总结

还有其他一些例子当前编译器编译不了，本文就暂时先举这几个例子。

C++ 静态反射，核心内容本篇几乎全覆盖到了，重要的是要理解反射的基本语法与标准元编程库，以及源码注入。静态反射和源码注入，对于产生式元编译来说，是非常强大的工具，能以此构建许多优秀的库或框架。但是目前仍不完善，还在发展中，语法也没有完全统一，要进标准还得不少时间。不过了解了这些概念，大家使用其他反射库应该不再是问题。

更多相关内容就随缘更新吧

注：在整理本书时，还没写后面几章，请留意公众号更新。

Part V

Concurrency

本 Part 是并发编程这一主题，主要包含并发/并行的相关内容，既然单独分出了一个部分出来，就表示这部分有一定难度。

这部分文章难度都在三到五星，对 C++20 *Coroutines* 进行了全方面讨论，还有线程相关并发组件也进行了一定讨论，除此之外，还讨论了 *Structured Concurrency* 这一未来特性。

相信通过学习本部分的内容，大家能够对 C++ 并发编程有更加深入的理解。

2023 年 5 月 14 日
里缪

Chapter 34

STRUCTURED CONCURRENCY IN C++

👤 Lucian Radu Teodorescu 📅 2022-04-14 💬 ★★★★

If one made a list of the paradigm changes that positively influenced software development, Structured Programming would probably be at the top. It revolutionised the software industry in two main ways: a) it taught us how to structure programs to better fit our mental abilities, and b) constituted a fundamental step in the development of high-level programming languages.

Work on structured programming began in the late 1950s with the introduction of the ALGOL 50 and ALGOL 60 programming languages, containing compound statements and code blocks, and allowing engineers to impose a structure on programs. In 1966, Böhm and Jacopini discovered what is now called *structured program theorem* [Böhm66]¹, proving that the principles of structured programming can be applied to solve all problems. Two years later, in 1968, Dijkstra published the highly influential article, ‘Go To Statement Considered Harmful’ [Dijkstra68]², arguing that we need to lay out the code in a way that the human mind can easily follow it (and that the GOTO statement works against this principle).

It is said that concurrency became a concern in the software industry in 1965, when the same Dijkstra wrote the article, ‘Solution of a problem in concurrent programming control’ [Dijkstra65]³. This article introduced the critical section (i.e., mutex), and provided a method of implementing it – the use of critical sections has remained an important technique for concurrent programming since its introduction. To reiterate the timeline, the critical section concept was introduced before the *structured programming theorem*, before the influential ‘Go To Statement Considered Harmful’ , and before 1968, when the term *Software Engineering* was coined (at a NATO conference [Naur69]⁴).

While concurrency is an old topic in the software industry, it appears that we still mainly handle concurrency in an unstructured way. Most of the programs that we write today rely heavily on raw threads and mutexes. And it’s not just the software that’s written this way; this is how we teach programmers

¹Corrado Böhm, Giuseppe Jacopini, ‘Flow Diagrams, Turing Machines and Languages With only Two Formation Rules’ , Communication of the ACM, May, 1966.

²Edgar Dijkstra, ‘Go To Considered Harmful’ , <https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf>, Communication of the ACM, March, 1968

³Edgar Dijkstra, ‘Solution of a problem in concurrent programming control’ , Communications of the ACM, September, 1965.

⁴Peter Naur, Brian Randell, Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October, 1968, 1969

concurrency. One can open virtually any book on concurrency, or look at most of the concurrency tutorials, and most probably the first chapters start with describing threads and mutexes (or any other form of synchronisation primitive). This is not anybody’s fault; it’s just the state of our industry (at least in C++).

This article aims to show that we can abandon the thread and mutex primitives for building concurrent abstractions, and start using properly structured concurrency with the senders/receivers framework [P2300R4]⁵ that may be included in C++23⁶. We look at the essential traits of structured programming, and we show that they can be found in the senders/receivers programming model.

34.1 Structured programming

The term *structured programming* is a bit overloaded. Different authors use the term to denote different things. Thus, it is important for us to define what we mean by it. We have 3 main sources for our definition of structured programming: a) the article by Böhm and Jacopini introducing the *structured program theorem* [Böhm66], Dijkstra’s article [Dijkstra68], and the book *Structured Programming* by Dahl, Dijkstra, and Hoare [Dahl72]⁷. The book contains references to the previous two articles, so, in a sense, is more complete.

Based on these sources, by *structured programming* we mean the following:

- use of abstractions as building blocks (both in code and data)
- recursive decomposition of the program as a method of creation/analysis of a program
- local reasoning helps understandability, and scope nesting is a good way of achieving local reasoning
- code blocks should have one entry and one exit point
- soundness and completeness: all programs can be safely written in a style that enables structured programming

In the following subsections, we will cover all these aspects of Structured Programming. Subsequently, we will use these as criteria for defining Structured Concurrency, and decide whether a concurrency approach is structured or not.

34.1.1 Use of abstractions

Abstraction is the reduction of a collection of entities to the essential parts while ignoring the non-essential ones. It’s the decision to concentrate on some of the similarities between elements of that set, discarding the differences between the elements. Variables are abstractions over ‘the current value’. Functions are abstractions that emphasise ‘what it does’ and discard the ‘how it does it’ .

As Brooks argues, software is essential complexity [Brooks95]⁸. That is, we cannot fit it into our heads. The only way to reason about it is to ignore parts of it. Abstractions are good ways to ignore the uninteresting details and keep in focus just the important parts. For example, one can easily understand and reason about something that ‘sorts an array’, but it’s much harder to understand the actual instructions that go into that sorting algorithm. And, even instructions are very high-level abstractions compared with the reality of electromagnetic forces and moving electrons.

⁵<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2300r4.html>

⁶里缪注：P2300 没能进入 C++23，应该能进入 C++26。

⁷O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, *Structured Programming*, Academic Press Ltd., 1972

⁸Frederick P. Brooks Jr., *The Mythical Man-Month* (anniversary ed.), Addison-Wesley Longman Publishing, 1995

Use of (good) abstractions is quintessential for structured programming. And to remind us of that, we have the following quote from Dijkstra [Dahl72]:

At this stage I find it hard to be very explicit about the abstraction, partly because it permeates the whole subject.

Out of all the abstractions possible in a programming language, the named abstractions (e.g., functions) are most useful. Not because they have a name associated with them, but because the engineer can create many of them and differentiate them by name. I.e., having the ability to add for loops in the code is good, but having the ability to create different functions that are capable of implementing any types of problems (including abstracting out for statements) is much more useful.

Using functions, the engineer can abstract out large parts of the program and express them with a simple syntactical unit (a function call). This means that functions play an important role in Structured Programming. When we say that abstractions are at the core of Structured Programming, we mean exactly this: we're able to abstract away large portions of the program.

Furthermore, it's worth noting that, in Structure Programming, one can find abstractions that would represent the whole program. Just think of the popular `main()` function.

34.1.2 Recursive decomposition of programs

In the *Structured Programming* book, Dijkstra spends a fair amount of time on problem decomposition, on how we're able to build software hierarchically. That is, we're capable of building programs by:

- recursively decomposing programs into parts, using a top-down approach
- making one decision at a time, the decision being local to the currently considered context
- ensuring that later decisions do not influence early decisions (for the majority of the decisions)

This gives us an (almost) linear process for solving software problems. Any process for solving software problems is good, but a linear process is much better. Software being essential complexity, one can imagine that we might have coupling between all parts of the system. If the system has N parts, then the amount of coupling would be in the order of $O(N^2)$. That is, a process that is focused on the interactions between these parts would have to be in the order of at least $O(N^2)$ steps. And yet, Structured Programming proposes us a linear process for solving problems.

The process advocated by Structured Programming is based on the *Divide Et Impera* strategy, and our brain is structured in such a way that allows us to easily cope with it.

One other key aspect of this method is that the sub-problems have the same general structure as the bigger problems. For example, let's assume that we have one function that calls two smaller functions. We can reason about all three functions in the same way: they all are abstractions, they all have the same shape (single entry, single exit point), they all can be further decomposed, etc. This will reduce the strain on our brains, and allows us to reuse the same type of solution in many places.

34.1.3 Local reasoning and nested scopes

To reduce the burden of understanding programs, Dijkstra remarks that we need to shorten the conceptual gap between the text of the program and how this is actually executed at runtime [Dahl72, Dijkstra68].

The closer the execution follows the program text, the easier it is for us to comprehend it. It is best if we can understand the consequences of running a piece of code by understanding just the preconditions and the corresponding instructions.

If we are following the pattern *preconditions + instructions \Rightarrow postconditions*, to properly have local reasoning (i.e., focus on the actual instructions), then we need the set of preconditions to be small. We cannot speak of local reasoning if the set of preconditions needs to contain information about everything else that happened before in the program. For example, if we want to analyse a block of code that sorts an array of numbers, we should not be concerned with how that array was generated; any parts of the sorting algorithm should not be coupled with other parts of the program.

Another way to look at local reasoning is by looking at encapsulation. As much as possible, all code blocks need to be fully encapsulated. Parts of a code block should not interact with the world outside the block. Dijkstra imagines that, in the ideal world, every part of the program would run on its dedicated (virtual) machine. This way, different parts will be completely independent.

In this context, the notion of scope is important. A lexical scope isolates the local concerns of a code block (lexically specified) from the rest of the blocks. If I declare a local variable in my block, no other block can interfere with my local variable. And again, the more stuff we have locally defined, the more local reasoning we're able to do.

34.1.4 Single entry, single exit point for code blocks

Looking at a sequence of regular instructions (i.e., without loops or alternatives) is easy. The preconditions of an instruction directly depend on the postconditions of the previous instruction. This is what Dijkstra calls **enumerative reasoning**. The conceptual gap between a sequence of instructions and the execution of those instructions in time is minimal.

If we want to treat code blocks or function calls as instructions, we should ensure that they share as many properties as possible with the simple instructions. One of these properties is **single entry, single exit point**. Every instruction, every block of code and every function should have one single entry point so that we can easily check whether the preconditions are met. Similarly, they should have one single exit point so that we analyse a single set of postconditions.

There is another advantage of using a single entry, single exit point strategy. The blocks and the function calls have the same shape as simple instructions. That allows us to apply the same type of reasoning to code blocks and to function calls, and permits us to have a simpler recursive decomposition.

34.1.5 Soundness and completeness

Having a single entry and a single exit point is a big restriction on the number of programs we can write. We need to make sure that this restriction does not impose a limit on the number of problems that can be solved with Structured Programming. We must have a sound method to ensure that we are able to build all programs with the restrictions imposed by our method.

The structured program theorem [Böhm66] proves that we can write all our programs using 3 simple control structures: sequence, selection and repetition. The Böhm and Jacopini paper has 3 major takeaways:

- ensuring that Structured Programming can be applied to all types of problems
- providing a set of primitives that can be used as building blocks for our programs

- providing alternative ways to visualise the programs – flowcharts

34.2 Concurrency with threads and synchronisation primitives

In the classic model of concurrency, one would use raw threads to express parallelism and then use synchronisation primitives to ensure that the threads do not interact in ways that break the correctness of the program. This model of organising concurrency is completely unstructured.

We don't have a general way of creating higher level abstractions that can represent parts of our concurrent program (and fully handle concurrency concerns inside it). Thus, we cannot use any such abstractions to decompose a program into subparts while keeping the concurrency constraints straight.

In this model, it's generally hard to do local reasoning. With synchronisation primitives, we almost always need to consider the rest of the program. We cannot simply look at one single function and draw any conclusion about what synchronisation we need.

As we don't have good general-purpose abstractions to encapsulate concurrency constraints, it is hard for us to discuss single-entry and single-exit points for such abstractions. Furthermore, as there is no general composable method for expressing concurrency, it's hard for us to discuss the soundness of the approach.

Synchronisation primitives act like GOTO commands, breaking local reasoning and encapsulation.

34.3 Concurrency with raw tasks

Let's now turn our attention to another model of dealing with concurrency. The one that is based on raw tasks (see [Teodorescu20]). We call a **task** an independent unit of work that is executed on one thread of execution. We call an **executor** something that dictates how and when tasks are executed. Those two concepts are enough for building concurrent systems.

We have proved before that a task-based system can be used to implement all concurrent problems [Teodorescu20]⁹ and that we can compose task-based system without losing correctness and efficiency [Teodorescu21]¹⁰. That is a step towards Structured Concurrency. However, task-based systems still don't fully have the characteristics we've taken from Structured Programming.

Let's first look at the ability to create concurrent abstractions. We may create abstractions that correspond to tasks, but it's hard to create abstractions of different sizes, spanning multiple threads. Taking tasks as they are, without any workarounds, doesn't fully encapsulate concurrency constraints. In this case, they also cannot be used for recursively decomposing concurrent programs.

Simple tasks are equivalent to functions, so they allow local reasoning just like Structured Programming does (that is, only if tasks are independent, as we said previously).

In [Teodorescu21], we introduced a technique to allow the decomposition of tasks. First, the concept of a task is extended to also contain a **continuation**. A continuation is a type-erased function that is executed whenever the task is completed. Most of the machinery that can be built on top of tasks can be made to work by using continuation. Secondly, the article introduced a trick that allows tasks to change their continuations while running. More precisely, to exchange the continuation of the current task with the continuation of another tasks (that can be executed in the future, or on a different thread). With this trick, we can make tasks represent concurrent work that span across multiple threads, with inner details.

⁹<https://accu.org/journals/overload/28/158/teodorescu/>

¹⁰<https://accu.org/journals/overload/29/162/teodorescu/>

If we apply this trick, then we lose the ability for local reasoning. The bigger abstraction (original task + continuation) is not specified in a one place; it is distributed between the start task, and all the tasks that are used to exchange the continuations (directly or indirectly) with this task. We lose lexical scoping and nesting properties.

With raw tasks, it's often the case that we **spawn** tasks and continue. This is the **fire-and-forget** style of creating concurrent work. Each time we do this, the **spawn** operation has one entry point, but two exit points: one that remains with the current thread, and one that follows the newly spawned task.

In conclusion, even if tasks systems allow us to achieve most of the goals we set up for a structured approach, we cannot achieve all the goals at the same time.

34.4 Concurrency with senders/receivers

There is a new model in C++ that can solve concurrency in a structured approach. This is the model proposed by [P2300R4], informally called senders/receivers. As we shall see, this model works well with all the important points considered to be ‘structured’ .

A sender is an entity that describes concurrent work. The concurrent work described by the sender has one entry point (starting the work) and one exit point, with three different alternatives: successful completion (possible with a value), completed with error (i.e., exception), or cancelled.

It is worth mentioning that senders just describe work; they don't encapsulate it. That would be the role of what P2300 calls **operation states**. However, users will rarely interact directly with operation states. These are hidden behind simple-to-use abstractions.

Listing 1 shows an example of using a sender to describe concurrent work. We specify that the work needs to be executed on `pool_scheduler` (e.g., an object identifying a pool of threads), we specify what work must happen, and we specify that if somehow the scheduler is cancelled to transform this into a specific error. While creating the sender object, no actual work happens; we just describe what the work looks like. The call to `sync_wait` starts the work and waits for it to complete, capturing the result of our concurrent work (or maybe forwarding the exception if any exception is thrown).

```

1 // Listing 1
2
3 using ex = std::execution;
4 int compute_something() {...}
5
6 ex::sender auto snd =
7     ex::schedule(pool_scheduler)
8     | ex::then(compute_something)
9     | ex::stopped_as_error(my_stopped_error)
10    ;
11 auto [r] = std::this_thread::sync_wait(std::move(snd))
12 .value();

```

For the purpose of this article, we define a **computation** as a chunk of work that can be executed on one or multiple threads, with one entry point and one exit point. The exit point can represent the three alternatives mentioned above: success, error and cancellation. Please note that having a multiple exit strategy doesn't

necessarily imply multiple exit points; this is similar to how function calls are considered to have a single exit point, even if they can exit either with a value or by throwing an exception.

A computation is a generalisation of a task. Everything that can be a task can also be a computation, but not the other way around.

The senders/receivers model allows us to describe any **computation** (i.e., any concurrent chunk of work) as one sender. A proof for this statement would be too lengthy to show here, and the reader can find it in [P2504R0]¹¹. This paper also shows the following:

- all programs can be described in terms of senders, without the need of synchronisation primitives
- any part of a program, that has one entry point and one exit point, can be described as a sender
- the entire program can be described as one sender
- any sufficiently large concurrent chunks of work can be decomposed into smaller chunks of work, which can be described with senders
- programs can be implemented using senders using maximum efficiency (under certain assumptions)

In summary, all concurrent single-entry single-exit chunks of works, i.e., computations, can be modelled with senders.

It is important to note that **computations** fully encapsulate concurrency concerns. **Computations** are to concurrent programming what functions are to Structured Programming. **Computations** are the concurrent version of functions.

The above paragraph represents the quintessence of this whole article.

34.4.1 Use of abstractions

Let's start our analysis of whether the senders/receivers model can be considered structured by looking at abstractions. In this model, the obvious abstraction is the computation, which can be represented by a sender. One can use senders to abstract out simple computations (smaller than typical tasks), to abstract out work that corresponds to a task, or to abstract out a chunk of work that may span multiple threads. The upper limit to how much work can be represented by a sender is the size of the program itself. Actually, the entire program can be represented by a single sender. See Listing 2 for how this may be done.

```

1 // Listing 2
2
3 using ex = std::execution;
4 ex::sender auto whole_program_sender() {...}
5 int main() {
6     auto [r] = std::this_thread::sync_wait
7         (whole_program_sender()).value();
8     return r;
9 }
```

¹¹<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2504r0.html>

34.4.2 Recursive decomposition of programs

As mentioned above, [P2504R0] proves that any program, or part of the program, can be recursively decomposed with the use of senders. Moreover, within a decomposition, the smaller parts can be made to look like the original part; they can be all senders.

Listing 3 shows an example on how a problem can be decomposed into two smaller parts, using senders to describe both the problem and the sub-problems.

```

1 // Listing 3
2 ex::sender auto do_process() {
3     ex::sender auto start
4         = ex::schedule(pool_scheduler);
5     ex::sender auto split_start = ex::split(start);
6     return ex::when_all(
7         split_start | ex::let_value(comp_first_half),
8         split_start | ex::let_value(comp_second_half)
9     );
10 }
```

34.4.3 Local reasoning and nested scopes

The reasoning can be local with the senders/receivers model. The definition of a sender completely describes the computation from beginning to end. The reasoning of all the aspects of that sender can be done locally, even if the sender contains multiple parts.

Let us look again at the code in Listing 3. The top-level computation contains the elements needed to reason about how the sub-computations start and finish. We’re able to inspect this code and draw the following conclusions:

- one entry and one exit for the entire process
- both halves start processing on the pool scheduler
- if one half ends with error/cancellation, then the other half is cancelled and the whole process ends with the original error/cancellation
- the process ends when both parts end

The idea of nested scopes was a major point in the design of senders/receivers. One can properly nest senders, and also nest operation states, and also receivers nest. Please see []¹² for more details.

34.4.4 Single entry, single exit point

By definition, senders have one entry point and one exit point. There are multiple alternatives if the work described by a sender may terminate, but essentially, we may say that there is only one exit point. This is similar to how a function call might exit either with a return value or an exception.

¹²<https://www.youtube.com/watch?v=xLboNIf7BTg>

While one can use the fire-and-forget style with senders, this is not the recommended way of using senders.

See also [Niebler21b]¹³ for a better illustration about how senders are fulfilling the single entry, single exit point requirement.

34.4.5 Soundness and completeness

In Structured Programming, the structured program theorem [Böhm66] provides the soundness and completeness guarantees for the method. For the senders/receivers model, as mentioned above, [P2504R0] shows that one can soundly use it to solve all possible concurrency problems.

If one is able to use senders to describe any chunk of concurrent work (with one entry and one exit point), and if we can decompose any concurrent programs in such work chunks, then we are covered.

The reader should note that, out of the box, P2300 doesn't provide primitives for representing repetitive computations or for type-erasing the senders; these are needed to be able to describe many concurrent chunks of work. However, the model is general enough to allow the user to create abstractions for these. In fact, there are libraries out there that already support these abstractions; one of the most known ones is [libunifex]¹⁴.

The senders/receivers model

The senders/receivers model proposed to C++ share many traits with the `async/await` model [Wikipedia]^a. The latter model was introduced in F# 2.0 in 2007 [Syme11]^b, and then spread to multiple programming languages (C#, Haskell, Python, etc.), eventually reaching C++ with the addition of coroutines in C++20. In F#, a type `Async<T>` represents an asynchronous computation (wording taken from [Syme11]). This would be analogous to the `Task<T>` coroutine type we discussed above. And, as senders are equivalent to these types, it means that the `Async<T>` from F# is an abstraction similar to a sender.

However, the senders/receivers model can be implemented more efficiently than coroutines or other similar type-erased abstractions.

Structured Concurrency is not something that can be done with senders/receivers only in C++. Other languages can support Structured Concurrency too; the P2300 senders/receivers model just ensures that we get the most performance out of it, as it avoids type-erasure and other performance penalties.

^a<https://en.wikipedia.org/wiki/Async/await>

^bDon Syme, Tomas Petricek, Dmitry Lomov. ‘The F# asynchronous programming model’, International Symposium on Practical Aspects of Declarative Languages, Springer, 2011

34.4.6 Bonus: coroutines

I felt it would be better not to include coroutines as one of the requirements needed for structured concurrency. We can write good concurrent code even without the use of coroutines. However, coroutines can help in writing good concurrent (and parallel) code.

Interestingly enough, in his part of the Structured Programming book [Dahl72], Dahl discusses coroutines, a SIMULA feature, as a way of organising programs. If a program uses only subroutines, then, at the

¹³<https://www.youtube.com/watch?v=6a0zzUBUNW4>

¹⁴<https://github.com/facebookexperimental/libunifex>

function-call level, the program is always organised hierarchically. Using coroutines might help when strict hierarchical organisation is not necessarily needed, and cooperative behaviour is more important.

The senders/receivers proposal guarantees good interoperability between senders and coroutines. According to the proposal, all awaitables are also senders (i.e., can be used in places where senders are used). Moreover, most of the senders (i.e., senders that yield a single value) can be easily made awaitables (by a simple annotation in the coroutine type).

Let us look at the example from Listing 4¹⁵. Here, `task<T>` is a coroutine type. The reference implementation associated with the P2300 paper provides an implementation for this abstraction [P2300RefImpl]¹⁶. `naive_par_fib` is a coroutine; it uses `co_return` twice and `co_await` once. The `res` variable inside the coroutine is a sender. We can `co_await` this sender, as shown in the body of our coroutine; thus, senders can be awaited. Towards the end of the example, we show how one can simply use the coroutine invocation to chain it to a sender, thus using it just as if it was a sender. Furthermore, the function passed to `let_value` needs to return a sender; we are returning a `task<uint64_t>` awaitable.

```

1 // Listing 4
2 template <ex::scheduler S>
3 task<uint64_t> naive_par_fib(S& sched, int n) {
4     if (n <= 2)
5         co_return 1;
6     else {
7         ex::sender auto start = ex::schedule(sched);
8         ex::sender auto res = ex::when_all(
9             start | ex::let_value([&] {
10                 return naive_par_fib(sched, n - 1);
11             }),
12             start | ex::let_value([&] {
13                 return naive_par_fib(sched, n - 2);
14             })
15         ) | ex::then(std::plus<uint64_t>());
16         co_return co_await std::move(res);
17     }
18 }
19 ex::sender auto snd =
20     naive_par_fib(sched, n)
21     | ex::then([](uint64_t res) {
22         std::cout << res << "\n";
23     });
24 std::this_thread::sync_wait(std::move(snd));

```

Using coroutines might make the user code easier to read. In this particular example, we also showed how coroutines can be used to obtain type-erasure of senders; `task<T>` acts like a type-erased sender that generates a value of type `T`.

¹⁵This is a bad implementation of a Fibonacci function, for multiple reasons. We use it here just to exemplify the interaction between components, not to showcase how one would implement a parallel version of Fibonacci

¹⁶https://github.com/bryceelbach/wg21_p2300_std_execution

34.5 Discussion

The current senders/receivers proposal, [P2300R4] doesn't aim at being a comprehensive framework for concurrent programming. For example, it lacks facilities for the following:

- type-erased senders
- a coroutine task type (the name is misleading, as it would encapsulate a computation)
- facilities for repeating computations
- facilities for enabled stream processing (reactive programming)
- serialiser-like abstractions
- etc.

However, the most important part of the proposal is the definition of the general framework needed to express concurrency. Then, all the facilities needed to represent all kinds of concurrent programs can be built on top of this framework. The proposal contains just a few such facilities (sender algorithms), but the users can create their own abstractions on top of this framework.

This article focuses on the framework, showing that it has the needed requirements for Structured Concurrency; we are not arguing that the facilities described in P2300 are enough.

Another important topic to discuss, especially in conjunction with concurrency, is performance. After all, the move towards more parallelism is driven by the performance needs of the applications. Our single-threaded processors are not fast enough, so we add more parallel processing to speed things up, and concurrency is needed to ensure the safety of the entire system.

In task-based systems, we put all the work into tasks. Usually, we type-erase these tasks to be able to pass them around. That means that we have an abstraction cost at the task level. This will make it impractical to use tasks for very small chunks of work. But, it is typically also impractical to use tasks with very big chunks of work, as this can lead to situations in which we do not properly fill up all the available cores. Thus, task-based systems have good performance in just a relatively narrow range, and users cannot fully control all performance aspects. Task-based systems can have good performance, but not always the best performance.

In contrast, the senders/receivers model doesn't imply any abstraction that might incur performance costs. The user is free to introduce type-erasure at any level that is best for the problem at hand. One can write large concurrent chunks of work without needing to pay the costs of type-erasure or memory allocation. Actually, most facilities provided in the current P2300 proposal do not incur any additional type-erasure or memory allocation costs.

The senders/receivers model allows one to model concurrency problems without any performance penalties.

34.6 conclusions

Structured Programming produced a revolution for the good in the software industry. Sadly, that revolution did not also happen in concurrent programming. To improve concurrent programming, we want to apply the core ideas of Structured Programming to concurrency.

After discussing the main ideas that shape Structured Programming, we briefly discussed that classic concurrent programming is not structured at all. We've shown that task-based programming is better, but still doesn't meet all the goals we set to enable Structured Concurrency.

The rest of this article showed how the senders/receivers model fully supports all the important ideas from Structured Programming, thus making concurrent programming with this model worthy to be called structured.

In concurrent programming, we have **computations** to play the role of functions in structured programming, and any **computation** can be described by a sender. With the help of [P2504R0], we argued that one can use senders to describe any concurrent program. We discussed here how senders can be used as abstractions to describe concurrent work at any level (from the entire program to small chunks of work). We've shown that this model lends itself well to recursive decomposition of concurrent programs and to local reasoning of abstractions. Likewise, we've also discussed how senders have the single entry, single exit shape and how one can build complex structures from simple primitives.

All these make the senders/receivers model match the criteria we set to achieve Structured Concurrency. When this proposal gets into the C++ standard, we will finally have a model to write safe, efficient and structured concurrent programs.

Chapter 35

Comparing Rust's and C++'s Concurrency Library

👤 Mara Bos 📅 2022-08-16 💬 ★★

The concurrency features that are included in the Rust standard library are quite similar to what was available in C++11: threads, atomics, mutexes, condition variables, and so on. In the past few years, however, C++ has gained quite a few new concurrency related features as part C++17 and C++20, with more proposals still coming in for future versions.

Let's take some time to review C++ concurrency features, discuss what their Rust equivalent could look like, and what it'd take to get there.

35.1 atomic_ref

P0019R8 introduced `std::atomic_ref` to C++. It's a type that allows you to use a non-atomic object as an atomic one. For example, you can create a `atomic_ref<int>` that references a regular `int`, allowing you the same functionality as if it were an `atomic<int>`.

While in C++ this needed a whole new type that duplicates most of the `atomic` interface, the equivalent Rust feature is a one-line function: `Atomic*::from_mut`. This function allows you to convert, for example, a `&mut u32` to a `&AtomicU32`, which is a form of aliasing that's perfectly sound in Rust.

The C++ `atomic_ref` type comes with safety requirements that you need to uphold manually. As long as you're using an `atomic_ref` to access an object, all access to that object must be through an `atomic_ref`. Accessing it directly when there's still an `atomic_ref` results in undefined behavior.

In Rust, however, this is already fully taken care of by the borrow checker. The compiler understands that by borrowing the `u32` mutably, nothing is allowed to access that `u32` directly until that borrow ends. The lifetime of the `&mut u32` that goes into the `from_mut` function is preserved as part of the `&AtomicU32` you get out of it. You can make as many copies of that `&AtomicU32` as you want, but the original borrow only ends once all copies of that reference are gone.

The `from_mut` function is currently unstable, but perhaps it's time we stabilize it.

35.2 Generic atomic type

In C++, the `std::atomic` is generic: you can have a `atomic<int>`, but also an `atomic<MyOwnStruct>`. In Rust, on the other hand, we only have specific atomic types: `AtomicU32`, `AtomicBool`, `AtomicUsize`, etc.

C++'s atomic type supports objects of any size, regardless of what the platform supports. It automatically falls back to a lock-based implementation for objects of a size that are not supported by the platform's native atomic operations. On the other hand, Rust only provides the types that are natively supported by the platform. If you're compiling for a platform that does not have 64 bit atomics, `AtomicU64` does not exist.

This has advantages and disadvantages. It means Rust code using `AtomicU64` might fail to compile for certain platforms, but it also means no performance related surprises when some types silently fall back to a very different implementation. It also means we can assume a `AtomicU64` is represented exactly the same as an `u64` in memory, allowing for functions like `AtomicU64::from_mut`.

Having a generic `Atomic<T>` in Rust that works for types of any size can be tricky. Without specialization, we can't make `Atomic<LargeThing>` include a `Mutex`, while not including it in `Atomic<SmallThing>`. What we could do, however, is to store the mutexes in a global `HashMap`, indexed by memory address. Then the `Atomic<T>` can be identical in size to a `T`, and use a `Mutex` from this global hash map when necessary.

This is exactly what the popular `atomic` crate does.

A proposal for adding such a universal `Atomic<T>` type to the Rust standard library would need to discuss whether it should be usable in `no_std` programs. A regular `HashMap` requires allocation, which isn't possible in `no_std` programs. A fixed size table could work for `no_std` programs, but might be undesirable for various reasons.

35.3 Compare-exchange with padding

P0528R3 changes how `compare_exchange` deals with padding. A compare exchange operation on a `atomic<TypeWithPadding>` used to compare the padding bits as well, but that turned out to be a bad idea. Nowadays, padding bits are no longer included in the comparison.

Since Rust currently only provides atomic types for integers, without any padding, this change is irrelevant for Rust.

However, a proposal for a `Atomic<T>` with a `compare_exchange` method would need to discuss how padding is handled, and should probably take input from this proposal.

35.4 Compare-exchange memory ordering

In C++11, the `compare_exchange` functions required the success memory ordering to be at least as strong as the failure ordering. A `compare_exchange(..., ..., memory_order_release, memory_order_acquire)` was not accepted. This requirement was copied verbatim to Rust's `compare_exchange` functions.

P0418R2 argued that this restriction should be lifted, which happened as part of C++17.

The same restriction is lifted as part of Rust 1.64, as part of rust-lang/rust#98383¹.

¹<https://github.com/rust-lang/rust/pull/98383>

35.5 `constexpr` Mutex constructor

C++'s `std::mutex` has a `constexpr` constructor, which means it can be constructed as part of constant evaluation at compile time. However, not all implementations actually provide this. For example, Microsoft's implementation of `std::mutex` doesn't include a `constexpr` constructor. So, relying on this is a bad idea for portable code.

Also, interestingly, C++'s `std::condition_variable` and `std::shared_mutex` don't provide a `constexpr` constructor at all.

Rust's original `Mutex` in Rust 1.0 did not include a `const fn new`. Combined with how Rust's strict requirements for static initialization, this made the `Mutex` quite annoying to use in a `static` variable.

This has been resolved in Rust 1.63.0² as part of rust-lang/rust#93740³: all of `Mutex::new`, `RwLock::new` and `Condvar::new` are now `const` functions.

35.6 Latches and barriers

P1135R6 introduced, among other things, `std::latch` and `std::barrier` to C++20. Both are types that allow waiting for several threads to reach a certain point. A latch is basically just a counter that gets decremented by each thread and allows you to wait for it to reach zero. It can only be used once. A barrier is a more advanced version of this idea that can be reused, and accepts a “completion function” to be automatically executed when the counter reaches zero.

Rust has had a similar `Barrier` type since 1.0. It was inspired by pthread (`pthread_barrier_t`) rather than C++.

Rust's (and pthread's) barrier is less flexible than what's now included in C++. It only has a “decrement and wait” operation (called `wait`), and lacks the “only wait”, “only decrement”, and “decrement and drop” functions that C++'s `std::barrier` comes with.

On the other hand, unlike C++, Rust's (and pthread's) “decrement and wait” operation assigns one thread to be the group leader. This is a (perhaps more flexible) alternative to a completion function.

The missing operations on the Rust version could easily be added at any point. All we need is a good proposal for the names of these new methods.:)

35.7 Semaphore

That same P1135R6 also added semaphores to C++20: `std::counting_semaphore` and `std::binary_semaphore`.

Rust does not have a general semaphore type, although it does equip every single thread with what's effectively a binary semaphore, through `thread::park` and `unpark`.

A semaphore can be easily constructed manually using a `Mutex<u32>` and a `Condvar`, but most operating systems allow for a more efficient and smaller implementation using a single `AtomicU32`. For example, through `futex()` on Linux and `WaitOnAddress()` on Windows. It depends on the operating system and its version which sizes of atomics can be used for these operations.

C++'s `counting_semaphore` is a template that takes an integer as argument to indicate how far we want to be able to count. For example, a `counting_semaphore<1000>` can count up to at least 1000, and will

²<https://blog.rust-lang.org/2022/08/11/Rust-1.63.0.html>

³<https://github.com/rust-lang/rust/issues/93740>

therefore be 16 bit or larger. The `binary_semaphore` type is just an alias for `counting_semaphore<1>`, and can be a single byte on some platforms.

In Rust, we're probably not quite ready for this kind of generic type any time soon. Rust's generics force a certain kind of consistency that puts some limitations on what we can do with constants as generic arguments.

We could have separate `Semaphore32`, `Semaphore64`, and so on, but that seems a bit overkill. Having `Semaphore<u32>`, `Semaphore<u64>` and perhaps even `Semaphore<bool>` could be possible, but is something we haven't done before in the standard library. Our atomic types are simply `AtomicU32`, `AtomicU64`, and so on.

As mentioned above, for our atomic types, we only provide the ones that are natively supported by the platform you're compiling for. If we were to apply the same philosophy to `Semaphore`, it wouldn't exist on platforms that don't have a `futex` or `WaitOnAddress` function, such as macOS. And if we had separate semaphore types for different sizes, some sizes wouldn't exist on (some versions of) Linux and various BSDs.

If we want a standard semaphore type in Rust, we'd first need some input on whether we actually need semaphores of different sizes, and what form of flexibility and portability would be necessary to make them useful. Perhaps we should go with just a single 32-bit `Semaphore` type that's always available (using a lock-based fallback), but any such proposal would have to include a detailed explanation of use cases and limitations.

35.8 Atomic wait and notify

The remaining new features that P1135R6 adds to C++20 are the atomic `wait` and `notify` functions.

These functions effectively directly expose Linux's `futex()` and Windows's `WaitOnAddress()` through a standard interface.

However, they are available on atomics of all sizes, on all platforms, regardless of what the operating system supports. Linux futexes are always 32 bit, but C++ allows for `atomic<uint64_t>::wait` just fine.

A way of doing this, is using something resembling a “parking lot⁴” : effectively a global `HashMap` that maps memory addresses to locks and queues. That means that a 32 bit wait operation on Linux could use the very fast futex based implementation, while the other sizes would use a very different implementation.

If we were to follow the philosophy of only providing the types and functions that are natively supported (like we do for the atomic types), we wouldn't provide such a fallback implementation. That'd mean we only have `AtomicU32::wait` (and `AtomicI32::wait`) on Linux, while all atomic types would include this `wait` method on Windows.

A proposal for `Atomic*::wait` and `Atomic*::notify` in Rust would need to include a discussion on whether a fall back to a global table is desirable in Rust or not.

35.9 jthread and stop_token

P0660R10 adds `std::jthread` and `std::stop_token` to C++20.

⁴<https://webkit.org/blog/6161/locking-in-webkit/>

If we ignore the `stop_token` for a second, `jthread` is basically just a regular `std::thread` that automatically gets `join()` ‘ed on destruction. This avoids accidentally detaching a thread and letting it run for longer than expected, which might happen with a regular `thread`. However, it also introduces a potential new pitfall: immediately destructing a `jthread` object will immediately join the thread, effectively removing any potential parallelism.

As of Rust 1.63.0⁵, we have scoped threads⁶ ([rust-lang/rust#93203](https://github.com/rust-lang/rust/issues/93203)⁷). Just like a `jthread`, a scoped thread is automatically joined. However, point before which they are joined is made explicit, and is a guarantee that can be relied upon for safety. The borrow checker even understands this guarantee, allowing you to safely borrow local variables in the scoped thread(s), as long as those variables outlive the scope.

In addition to automatically joining, a main feature of `jthreads` is their `stop_token` and corresponding `stop_source`. One can call `request_stop()` on a `stop_source` to make the corresponding `stop_requested()` method on `stop_token` return `true`. This can be used to nicely ask the thread to please stop, and is automatically done in the destructor of `jthread` before joining. It’s up to the code of the thread to actually check the token and stop if it was set.

So far, it almost looks like a plain `AtomicBool`.

Where things get very different is the `stop_callback` type. This type allows registering a callback, a “stop function”, to be registered with a stop token. Requesting a stop using the corresponding stop source will execute this function. Effectively, a thread can use this to let others know how to stop or cancel its work.

In Rust, we could easily add the `AtomicBool`-like functionality to the `Scope` object of `thread::scope`. A simple `is_finished(&self) -> bool` or `stop_requested(&self) -> bool` that indicates whether the main `scope` function is finished might suffice. Maybe combined with a `request_stop(&self)` method to request it from anywhere.

The `stop_callback` feature is more complicated, and any Rust equivalent would probably need a detailed proposal discussing its interface, use cases and limitations.

35.10 Atomic floats

P0020R6 adds support for atomic floating point addition and subtraction to C++20.

It’d be easy to add a `AtomicF32` or `AtomicF64` to Rust as well, but it seems that the only platforms that natively support atomic floating point operations are some GPUs that are not supported by Rust (yet?).

A proposal to add these types to Rust would have to present some compelling use cases.

35.11 Atomic per byte memcpy

Currently, it’s not possible to efficiently implement sequence locks⁸ in Rust or C++ that abides by all the rules of the memory model.

P1478R7 proposes to add `atomic_load_per_byte_memcpy` and `atomic_store_per_byte_memcpy` to a future version of C++ to solve this issue.

For Rust, I wrote a proposal to expose the functionality through a `AtomicPerByte<T>` type: RFC 3301⁹.

⁵<https://blog.rust-lang.org/2022/08/11/Rust-1.63.0.html>

⁶<https://doc.rust-lang.org/stable/std/thread/fn.scope.html>

⁷<https://github.com/rust-lang/rust/issues/93203>

⁸<https://en.wikipedia.org/wiki/Seqlock>

⁹<https://github.com/rust-lang/rfcs/pull/3301>

35.12 Atomic shared_ptr

P0718R2 added specializations for `atomic<shared_ptr>` and `atomic<weak_ptr>` to C++20.

Reference counted pointers (`shared_ptr` in C++, `Arc` in Rust) are quite commonly used for concurrent lock-free data structures. The `atomic<shared_ptr>` specialization makes it easier to do this correctly, by handling the reference count properly.

In Rust, we could add equivalent `AtomicArc<T>` and `AtomicWeak<T>` types. (Although `AtomicArc` sounds a bit weird maybe, considering the `A` of `Arc` already stands for “atomic” . . :))

However, C++’s `shared_ptr<T>` is nullable, while in Rust that requires a `Option<Arc<T>>`. It’s not immediately clear whether `AtomicArc<T>` should be nullable, or whether we also have a `AtomicOptionArc<T>`.

The popular arc-swap crate¹⁰ already provides all these variants in Rust, but, as far as I know, there hasn’t been any proposal yet to add anything similar to the standard library.

35.13 synchronized_value

P0290R2 was not accepted, but proposed a type called `synchronized_value<T>` which combines a `mutex` with a `T`. Even though it wasn’t accepted at that time into C++, it’s an interesting proposal, because `synchronized_value<T>` is pretty much exactly what a `Mutex<T>` is in Rust.

In C++, a `std::mutex` does not contain the data it protects, nor does it even know what it is protecting at all. This means that it is the responsibility of the user to remember which data is protected and by which mutex, and ensure the right mutex is locked every time “protected” data is accessed.

Rust’s `Mutex<T>` design with a `MutexGuard` behaving like a (mutable) reference to `T` allows for much more safety, while still allowing for a `Mutex<()>` in cases where you need only a mutex, without any data directly attached to it. The proposal for `synchronized_value<T>` was an attempt at adding this pattern to C++, but used closures instead of a mutex guard, since C++ doesn’t track lifetimes.

35.14 Conclusion

It seems to me that C++ can continue to be a source of inspiration for Rust, although we should take care not to copy-paste ideas directly. As we’ve seen with `Mutex<T>`, scoped threads, `Atomic*::from_mut` and others, things can often take a very different (often more ergonomic) shape in Rust while providing the same functionality.

Providing the exact same functionality as C++ shouldn’t be a primary goal. The goal should be to provide exactly what the Rust ecosystem needs from the language and standard library, which might be different than what C++ users need from their language.

If you have concurrency needs from the Rust standard library that we currently don’t fulfill, I’d love to hear from you, regardless of whether it’s something that’s already solved in another language or not.

¹⁰<https://docs.rs/arc-swap/>

Chapter 36

C++ 20 concurrency

👤 Gajendra Gulgulia 📅 2022-01-06 💬 ★★★

36.1 Part 1: synchronized output stream

Most of C++ developers who have been using concurrency features are well aware of the problems of `std::ostream` or `std::cout` with multi threading. The outputs are almost all the time undeterministic and worse they are always entangled with each other in ways which makes `std::cout` unusable in concurrent applications.

36.1.1 The Problem

Consider the following program:

```
1 #include <thread>
2 #include <iostream>
3
4 void square_and_print(const double num){
5     std::cout<<"square of number "<< num << ":" " << num*num << "\n";
6 }
7
8 int main(){
9     std::jthread t1{square_and_print, 3.14};
10    std::jthread t2{square_and_print, 20.0};
11    std::jthread t3{square_and_print, 1};
12    std::jthread t4{square_and_print, 5};
13    return 0;
14 }
```

The output I see with gcc 11.2 compiler:

```
square of number square of number square of number 120: : 1
400
square of number 5: 25
3.14: 9.8596
```

This looks quite messed up and garbled with each other. On a second run of the same program, I get the following output:

```

square of number square of number 3.14: 9.8596
20: 400
square of number 5: 25
square of number 1: 1

```

Only last two console statements look correctly printed which also happened due to pure luck. In reality the order of each string appearing on the console is completely nondeterministic and the strings can appear in any order.

36.1.2 The solution

C++20 finally has a solution for this : std::osyncstream . I' ll not go into the details in the cpp reference and directly show how to use std::osyncstream to get a synchrnoized output in the code below

```

1 #include <iostream>
2 #include <thread>
3 #include <syncstream> // Note this
4
5 void square_and_print(const double num){
6     /* create an object of type std::osyncstream
7         and initialize it with std::cout */
8     std::osyncstream syncout{std::cout};
9     /* and simply use the sync stream object in place of std::cout */
10    syncout << "square of number " << num << ":" << num*num << "\n";
11 }
12
13 output
14 -----
15 square of number 3.14: 9.8596
16 square of number 5: 25
17 square of number 1: 1
18 square of number 20: 400

```

Now the outputs are free from data races and not garbled with that from other threads. The order of output still depends on the threads scheduled first to execute (mostly by the operating system).

36.1.3 Underlying mechanism

`std::osyncstream` on every instantiation of the temporary syncout creates a so called **synchronized output buffer**. The cpp refernece page says the following about this synchronized output buffer:

…all output made to the same final destination buffer (std::cout in the examples above) will be free of data races and will not be interleaved or garbled in any way, as long as every write to the that final destination buffer is made through (possibly different) instances of `std::basic_osyncstream`

This simply means that even if multiple instances of `std::osyncstream` are used in a method, the outputs are still guaranteed to be synchronized to a final destination buffer.

There's however another detail that is important to note. In the proposal¹ for `std::osyncstream`, it is mentioned that the temporary object has its own buffer that is actually synchronized and onto which the outputs are written without any data races. When the destructor of the temporary `std::osyncstream` is called, in our case the object `syncout`, the contents of the buffer is written to another buffer which finally gets printed.

The consequence is that `std::endl` or `\n` doesn't flush the output like it does in `std::ostream` (or `std::cout`)

```
syncout << "some string stream " << std::endl; //std::endl DOES NOT
//flush output
```

To be able to flush the output before the destructor of `std::osyncstream` on a temporary object is executed, we can therefore use `std::flush_emit` which surprisingly prevents all interleaved outputs in the presence of mixed stream outputs (`std::ostream` and `std::osyncstream`)

```
syncout << "some string stream " << std::flush_emit; //guarantees
//output is flushed everytime
```

36.1.4 Synchronized output stream for files

With `std::osyncstream`, we can use a file output stream, `std::ofstream` to ensure data race free output buffer stream which is finally dumped to a file when the `std::osyncstream` object is destroyed

```
1 #include <thread>
2 #include <fstream>
3 #include <syncstream>
4 #include <functional> //std::ref
5 void square_and_file_write(int num, std::ostream& streamBuff){
6
7     std::osyncstream syncout{streamBuff};
8     syncout << "square of num is: " << num*num << "\n"
9     << std::flush_emit;
10 }
11 int main(){
12     std::ofstream filestream{"test.txt"};
13     std::thread t1{square_and_file_write, 1, std::ref(filestream)};
14     std::thread t2{square_and_file_write, 2, std::ref(filestream)};
15     t1.join();
16     t2.join();
17     return 0;
18 }
```

Here the same `filestream` is being passed as a parameter to the function `square_and_file_write` but each `filestream` has its own `std::osyncstream` buffer which means that the buffers are not written to by different threads and that the stream buffers have no race conditions. Finally to ensure that each thread prints a new

¹<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0053r1.html>

line character, we must use either `std::endl` or "`\n`" escape sequence since `std::flush_emit` only flushes the output from `std::osyncstream` and doesn't print a new line character.

36.1.5 Conclusion

`std::osyncstream` is a useful and much needed feature in C++ that fills the gap in multi-threading and stream buffers. This will, in my opinion make logging features seamlessly easy in C++ multi-threaded applications. Stay tuned for more articles on C++20's concurrency feature.

36.2 Part 2: jthreads

In this part of the issue, I'll discuss about the new `std::jthread` that helps us avoid the boilerplate code for joining the conventional `std::thread` in the first section. In the end, I'll also mention about the `std::swap` algorithm's specialization introduced in C++20 to swap the underlying thread handles associated with `std::jthread`.

36.2.1 std::jthread

`std::jthread` is arguably the most awaited and needed concurrency feature in C++. Before its existence, the users had only access to the `std::thread`'s interface and had to rely on a boilerplate code to ensure a proper resource management with `std::thread`.

36.2.1.1 The Problem

For example, a common task that the client of `std::thread` had to perform was joining the thread hence ensuring also a barrier or synchronization before the thread object goes out of scope to free up the resources (hardware and operating system) consumed by the thread. For e.g. in the example below

```

1 #include <thread>
2 void do_task(int param){
3     std::cout << "doing task with " << param << " from thread: "
4             << std::this_thread::get_id() << std::endl;
5
6     /* do something with param1 */
7 }
8 int main(){
9     std::thread t1{do_task, 1};
10    std::thread t2{do_task, 2};
11    t2.detach();
12    //boilerplate code to manage thread resource
13    if(t1.joinable()){ t1.join() } //joinable returns true, barrier
14    if(t2.joinable()){ t2.join() } //joinable returns false
15 }
```

If an instance of `std::thread` is used as a part of the object, then the resource management (according to RAI) is left to destructor:

```

1 #include <thread>
2 #include <functional>
3 class ParallelTaskWrapper{
4     private:
5         std::functional<void(int)> task;
6         std::thread taskThread;
7     public:
8         ParallelTaskWrapper(std::functional<void(int)> t_, int param):
9             task{std::move(t_)}
10        {
11             taskThread = std::thread{task, param};
12        }
13        /* boilerplate code to manage thread resource in destructor */
14        ~ParallelTaskWrapper(){
15            if(taskThread.joinable()){
16                taskThread.join()
17            }
18        };

```

36.2.1.2 The Solution

To get rid of the boilerplate, in C++20 we can use `std::jthread` and refactor the code from pre-C++20 to replace `jthread` where ever `std::thread` has been used. The destructor of `std::jthread` does the resource management for us and joins the thread if it is joinable before the thread object goes out of scope. `std::jthread` is also available in the header `thread` header. The first line of cpp reference page on `std::jthread` says the following:

The class `jthread` represents a single thread of execution. It has the same general behavior as `std::thread`, except that `jthread` automatically rejoins on destruction ...

Just with this much knowledge in sight, the last two examples can be refactored. Notice in the two examples below, how the `std::thread` has been replaced with `std::jthread`.

Example 1

```

1 #include <thread>
2
3 void do_task(int param){
4     std::cout << "doing task with " << param << " from thread: "
5             << std::this_thread::get_id() << std::endl;
6
7     /* do something with param1 */
8 }
9
10 int main(){
11     std::jthread t1{do_task, 1};

```

```

12     std::jhread t2{do_task, 2};
13     t2.detach();
14
15 }
```

Example 2

```

1 #include <thread>
2 #include <functional>
3
4 class ParallelTaskWrapper{
5     private:
6         std::functional<void(int)> task;
7         std::jthread taskThread;
8
9     public:
10     ...
11     ~ParallelTaskWrapper(){
12         /* no more boiler plate code in destructor*/
13     }
14 };
```

36.2.2 Swapping two std::jthread

Apart from the other additions in `std::jthreads`, C++20 introduced a specialization for swap algorithm(`std::swap`) to simply exchange the underlying handles representing two threads of execution. `swap` is also a member function of `std::jthread` and both have exactly the same functionality . Note the example below

```

1 #include <thread>
2 #include <chrono>
3
4 using namespace std::chrono_literals;
5 void foo(){ std::this_thread::sleep_for(1s); }
6 void bar(){ std::this_thread::sleep_for(1s); }
7
8 int main()
9 {
10
11     std::jthread t1(foo);
12     std::jthread t2(bar);
13
14     std::cout << "thread 1 id: " << t1.get_id() << '\n'
15             << "thread 2 id: " << t2.get_id() << '\n';
16
17     //using std::swap algorithm on std::thread
18     std::swap(t1, t2);
```

```

19
20     std::cout << "after std::swap(t1, t2):" << '\n'
21             << "thread 1 id: " << t1.get_id() << '\n'
22             << "thread 2 id: " << t2.get_id() << '\n';
23
24     //using member function std::jthread::swap
25     t1.swap(t2);
26
27     std::cout << "after t1.swap(t2):" << '\n'
28             << "thread 1 id: " << t1.get_id() << '\n'
29             << "thread 2 id: " << t2.get_id() << '\n';
30
31
32 }
```

36.2.3 Conclusion

`std::jthread` offers the same functionalities (and more) as compared to `std::thread` but is a RAII object, just like the smart pointers introduced in C++11, and does the necessary resource management for us by joining the thread, if possible before the `std::jthread` object is destructed. Unlike smart pointers, it is really easy to replace `std::thread` with `std::jthread` and refactor the code to more modern C++. In the next article, I'll discuss about the stop token functions introduced with `std::jthread` to interrupt an executing thread. So stay tuned for more modern C++.

36.3 Part 3: request_stop and stop_token for std::jthread

In this part, we explore the latest feature of C++20's `std::jthread` that allows one to signal a stop or cancellation to an already executing thread in certain situation. Quoting the first line again from cpp reference:

the class `jthread` represents a single thread of execution..., and can be cancelled/stopped in certain situations.

To keep my article readable, I'll omit `std::` and all occurrences of library constructs imply that they belong to C++'s `std` namespace unless explicitly stated otherwise.

36.3.1 Introduction: Two ways to cooperatively stop the thread

`jthread` provides a **cooperative** means to stop a thread of execution which implies that the threads cannot be interrupted² or killed³ but can only be signaled to stop. There are two ways to cooperatively stop the thread and both the methods make use of a **shared stop-state** of type `std::stop_source`.

With the help of shared stop-state (`stop_source`) and a `std::stop_token` a check can be performed whether or not a stop request has been made. If it has been made, then the method that the thread executes can return

²Explanation about interruption: Page 4 of <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0660r10.pdf>

³Killing a thread may cause an executing thread to interrupt in the middle of its job which may leak resources. Also killing a thread usually requires dealing with OS kernels.

immediately, and the responsibility to provide the implementation to return on a stop request falls on the programmer. But the mechanisms as to how the `stop_token` can be signaled to stop may vary and can be grouped broadly into two categories:

1. by relying on `jthread` ‘s internal shared stop-state.
2. by employing an external shared stop-state.

In this article, I’ll explain how to cooperatively signal a stop/cancellation to an executing `jthread` by relying on `jthread` ‘s internal shared stop-state and making use of `jthread::request_stop` or `jthread::jthread` (or `jthread` ‘s destructor). Therefore in this article I’ll not explain about `std::stop_source` since it will be managed internally by `jthread`.

36.3.2 Using `std::jthread` internal stop-state

First line of the second paragraph of cpp reference page says the following about `jthread`’s internal stop-state:

`jthread` logically holds an internal private member of type `stop_source`, which maintains a shared stop-state.

When a task in the form of function is passed to `jthread` ‘s constructor, the internal shared stop-state ‘s stop token (`std::stop_token`) is passed to the function, if the function accepts a `stop_token` as a first parameter. Therefore to be able to react to `jthread::request_stop()`, the function that the `jthread` executes must admit a first parameter as an `stop_token`. This is exactly what cpp reference page says about a task being assigned to a `jthread`.

The `jthread` constructor accepts a function that takes a `std::stop_token` as its first argument, which will be passed in by the `jthread` from its internal `stop_source`.

36.3.2.1 A first example without `std::stop_token` as function’s first parameter

Consider the following code where `jthread::request_stop` method is being executed but the thread never stops even though the method `request_stop` is called before the thread `t1` finishes before it could finish it’s approximately 6 seconds of execution time.

```

1 #include <iostream>
2 #include <thread>
3 #include <syncstream>
4
5 void foo(){
6     std::osyncstream syncout{std::cout};
7     //execute for approx 6 seconds
8     for(int secs=0; secs<6; ++secs){
9         std::this_thread::sleep_for(1s);
10        syncout <<"from foo: " << secs+1
11                << " seconds elapsed\n"

```

```

12         << std::flush_emit;
13     }
14 }
15
16 void stop_foo(){
17     std::jthread t1{foo};           //1:
18     std::this_thread::sleep_for(2s); //2:
19
20     for(int secs=0; secs<2; ++secs){
21         std::this_thread::sleep_for(1s);
22         std::cout << "stop_foo ran for << secs+1 << " seconds\n";
23     }
24     t1.request_stop();           //3:
25 }
26
27 int main() { stop_foo(); }
```

In the above code snippet, in the function `stop_foo` following points marked as 1, 2 and 3 in the function `stop_foo` are to be noted:

1. first the task foo is launched on thread t1 which can run for approx 6 seconds
2. next the main thread is put to sleep for 2 seconds and allowed to further run for 2 seconds
3. finally t1 is signaled to stop while it is still in its execution (hopefully in approximately 4th second).

The output on console looks (as expected) like below:

```

from foo: 1 seconds elapsed
from foo: 2 seconds elapsed
from stop_foo: 1 seconds elapsed
from foo: 3 seconds elapsed
from stop_foo: 2 seconds elapsed
from foo: 4 seconds elapsed
from foo: 5 seconds elapsed
from foo: 6 seconds elapsed
```

This is expected since the function `foo` doesn't take `stop_token` as its first parameter and therefore calls to `t1.request_stop()` doesn't affect `foo` being executed in `t1`.

36.3.2.2 A first example WITH std::stop_token as function's first parameter

Now lets have a same function but modify it slightly:

1. pass `stop_token` as the first parameter to the function `foo`
2. modify the body of `foo` to query if stop signal has been requested on the stop token

The modified part of the code is in bold and the output with the new modifications are:

```

1 void foo(std::stop_token token){
2     ...
3     for(int secs=0; secs<6; ++secs){
4         ...
5         //query from time to time if stop has been requested
6         if (token.stop_requested()) {
7             std::cout << "foo requested to stop\n";
8             return;
9         }
10    }
11 }
12
13 output
14 -----
15 from foo: 1 seconds elapsed
16 from foo: 2 seconds elapsed
17 from stop_foo: 1 seconds elapsed
18 from foo: 3 seconds elapsed
19 from stop_foo: 2 seconds elapsed
20 from foo: 4 seconds elapsed
21 foo requested to stop

```

Note in the output that `foo` printed only 4 counts instead of early 6 counts and the last print was from within the `if`-condition which was triggered approximately after 4 seconds from main thread.

With the new modifications in `foo`, the main thread is able to send a signal to the `stop_token` in thread `t1` and when received in `foo` the body of `if` condition returns from `foo`, or in other words, the `foo` cooperates and returns early. The working example can be accessed from this link⁴ in compiler explorer .

36.3.2.3 stop signal as a result of destructor of jthread being executed

Apart from manually signalling the thread to stop, the robust RAII implemented in `jthread` ‘s destructor in that **if the thread is joinable** the destructor sends a stop signal to the function before it can join.

In the case of `foo` it is programmed to run approximately 6 seconds but the thread that calls `foo` within the funciton `stop_foo` is “alive” only for approximately 4 seconds. Therefore even if `t1.request_stop()` is not executed, `t1` goes out of scope after 4 seconds and the destructor is executed causing `foo` to stop early. The modified code for `stop_foo` is below where `t1.request_stop()` is simply commented out

```

1 void stop_foo(){
2     ...
3
4     for(int secs=0; secs<2; ++secs){
5         ...
6         ...

```

⁴<https://godbolt.org/z/KnGcWvch>

```

7     }
8     //t1.request_stop();
9     //t1 goes out of scope after ~ 4 seconds
10    // if(t1.joinable()) then t1.~jthread sends a stop signal to foo
11 }

```

And the output remains same as in section 2.2. The final example can be seen here in compiler explorer⁵.

36.3.2.4 Attaching a stop_callback for a stop_token within the function

According to the cpp reference, the `std::stop_callback` is a RAII object type that registers a callback within a function whenever the thread that executes the function receives a `request_stop` (as we saw in section 2.2) or is going out of scope (as shown in section 2.3). Stop callbacks can be registered anywhere but it has to be associated with the same shared stop-state of a working thread. The stop callback gets invoked either from the thread that invokes `request_stop` either explicitly or due to `jthread` ‘s destruction if the stop has not yet been requested; or if a stop has already been requested, then it is invoked from the thread where it was constructed.

Lets have a first glimpse of how this works by registering a `stop_callback` within the `foo` function (the task that is asynchronously launched):

```

1 void foo(std::stop_token token){
2     ...
3
4     //register stop callback
5     std::stop_callback cb{token, [&syncout]{
6         syncout << "stop callback executed from
7             << std::this_thread::get_id()
8             << "\n" << std::flush_emit
9     }};
10
11    for(int secs=0; secs<6; ++secs){
12        ...
13    }
14 }
15 }
16
17 void stop_foo(){
18     std::osyncstream syncout{std::cout};
19
20     //get the thread id of stop_foo function
21     syncout << "stop foo in thread: "
22         << std::this_thread::get_id()
23         << "\n" << std::flush_emit;

```

⁵<https://godbolt.org/z/s7zfYhEzd>

```

24     ...
25     for(int secs=0; secs<6; ++secs){...}
26
27     //launch another thread and request stop from there
28     std::jthread stop_req_thread{
29         [&t1, &syncout]{
30             syncout << "stop_req_thread thread id: "
31                 << std::this_thread::get_id()
32                 << "\n" << std::flush_emit;
33             t1.request_stop();
34         };
35
36     int main() { stop_foo();}

```

In the above code snippet, the callback is registered within the function `foo`, which is started on the thread `t1` as previously. What is interesting is that another thread `stop_req_thread` is launched which captures `t1` in its lambda-capture and inside this thread the `request_stop` on `t1` is executed.

This causes the `stop_callback` to be executed from the thread `stop_req_thread`, even though it was constructed on thread `t1`, as can be observed from the output

```

stop foo in thread: 139771773687616
thread id of foo: 139853637019392
from foo: 1 seconds elapsed
from foo: 2 seconds elapsed
from stop_foo: 0 seconds elapsed
from foo: 3 seconds elapsed
from stop_foo: 1 seconds elapsed
stop_req_thread id: 139771765290752
foo requested to stop
stop callback executed from 139771765290752

```

The entire example can be accessed in this link⁶ from compiler explorer.

36.3.3 Summary

There are two ways to use shared stop-state: (1) by using the `jthread` ‘s own shared stop-state or (2) by explicitly creating a shared stop-state and tying it to a `jthread` ‘s stop token. In this article we looked at the first method.

In this method, we looked at the mechanism by which the `stop_token` associated with the `jthread` ‘s shared stop state receives a stop signal viz: (1) by explicit calls to `jthread::request_stop` or when the destructor of `jthread` gets executed.

In each of the case the task being executed must have `stop_token` as a first optional parameter that gets tied to `jthread`’s internal shared stop-state when the constructor of `jthread` receives the task and any other

⁶<https://godbolt.org/z/16vPoGhfY>

arguments of the task (in our case `foo` had no arguments, but feel free to provide parameters to `foo` and pass valid arguments to `foo` ‘s parameters).

Finally a `stop_callback` could be registered with `jthread` ‘s shared stop-state which is very versatile in terms of RAI^I, preventing race conditions. We only looked at a simple example in this article with `stop_callback` but in the 4th issue of C++20’s cocurrency features, I’ll dive deeper into the usage of `stop_callbacks` and the second method of cooperatively stopping a `jthread`.

Chapter 37

co_resource<T>: An RAII coroutine

👤 vector<bool> 📅 2021-12-30 🏷★★★

Python has a syntactic construct known as a with statement:

```
with open('some-file.txt') as f:  
    ... # code goes here
```

Code within the `with` block will execute as normal, but if the `with` block is exited for any reason, then some cleanup occurs. It works like this:

1. The operand to `with` is captured as a context manager `c`.
2. Upon entering the `with` block, `c.__enter__()` is called, and the result assigned to variable named with the `as` specifier (which may be omitted).
3. Upon exiting the `with` block, `c.__exit__(...)` is called (with some arguments pertaining to exception handling, but they are not important here).

The `__enter__` and `__exit__` methods can perform arbitrary actions. You can think of this as a way to capture a `finally` block in an external variable. `with` supports a variety of objects:

```
# Files: Automatically calls `close()` when finished  
with open(filepath) as f:  
    ...  
  
# Locks: Calls `acquire` on __enter__, and `release` on __exit__  
with some_mutex:  
    ...  
  
# Databases: Begins a transaction on __enter__, and commits or rolls  
#   back on __exit__, depending on whether an exception occurred.  
with some_db_client:  
    ...
```

C++ developers will see this and understand that we have destructors to manage these kinds of things: Regardless of how the scope is exited (`throw`, `return`, or even `goto`), the destructors of our local variables will be invoked deterministically.

But here's a neat trick: Python's standard library contains a decorator called `contextlib.contextmanager` (Refer¹). This function decorator transforms any (generator) coroutine into a context manager object that can be used as the operand to a `with` statement:

```
from contextlib import contextmanager

@contextmanager
def printing(msg: str):
    print(f'Entering: {str}')
    # Yield a value, making this into a coroutine function
    yield 1729
    print(f'Exiting: {str}')

with printing('coro context example') as val:
    print(f'Got value: {val}'')
```

This program, when executed, will print:

```
Entering: coro context example
Got value: 1729
Exiting: coro context example
```

It works like this:

1. When we call `printing()`, the `@contextmanager` internals will return a special context manager object that initializes and manages a new generator coroutine created by invoking the decorated coroutine function.
2. When the `with` statement calls `__enter__` on the `@contextmanager`-generated value, the context manager performs the initial send (initial resume) to get the coroutine started.
3. When the coroutine execution hits the `yield` expression, the coroutine suspends and control passes back to the `__enter__` method, which receives the yielded value. The yielded value is then returned from `__enter__` and bound to the `with-as` variable `val`.
4. We run the inner block and print `val`
5. When we exit the with block, the `__exit__` method in the `@contextmanager` object resumes the coroutine again, expecting that it will return without yielding further.
6. The coroutine reaches the end of the function, and the `__exit__` method resumes again.

Importantly: A coroutine execution managed by a `@contextmanager` must evaluate a single `yield` expression exactly once.

Pedantic aside: A Python function that `yields` is technically a generator, which is a Python coroutine of a specific flavor, and a “coroutine” usually refers to an `async def` function that `awaits`, but this distinction is unimportant for our purposes.

¹<https://docs.python.org/3/library/contextlib.html#contextlib.contextmanager>

37.1 Language Envy

I've been writing a lot of C++ and Python code recently, and the `@contextmanager` decorator is absolutely one of my favorite things in Python, and I've found myself on multiple occasions wondered if I could get the same thing in C++ using C++20's coroutines.

Skeptics may hear this and think this is pointless: We have destructors and RAII in C++, which allow us to already perform arbitrary deterministic cleanup, so why would we need something like this?

One important benefit of `@contextmanager` over a constructor-destructor (or even a `try/finally`) pair is "locality of code":

```

1 struct file_scope {
2     std::FILE* file;
3
4     file_scope(const char* path)
5         : file(std::fopen(path, "rb")) {}
6
7     ~file_scope() {
8         std::fclose(file);
9     }
10};

```

This is a fairly small example, but illustrates that there's some amount of unfortunate separation between the "initialization" and "cleanup" code. A more complex example might have the constructor and destructor implemented out-of-line.

Another unfortunate effect would relate to storage of supporting state. Here's a simple example for a reentrant database transaction guard:

```

1 struct recursive_transaction_guard {
2     database_ref db;
3     int n_exceptions = std::uncaught_exceptions();
4     bool already_in_transaction = db.in_transaction();
5
6     recursive_transaction_guard(database_ref db) : db(db) {
7         if (!already_in_transaction) {
8             db.begin();
9         }
10    }
11
12    ~recursive_transaction_guard() {
13        if (already_in_transaction) {
14            // Do nothing
15            return;
16        }
17        if (std::uncaught_exceptions() > n_exceptions) {

```

```

18     // We're exiting with an error
19     db.rollback();
20   } else {
21     // We're okay
22     db.commit();
23   }
24 }
25 };

```

Here the associated state to track the database and exception state is stored as additional member variables. It isn't intractible, but it feels a bit more clunky than an equivalent in Python with `@contextmanager`:

```

@contextmanager
def recursive_transaction(db: Database):
    if db.in_transaction:
        # Do nothing
        yield
        return

    db.begin()
    try:
        yield
        db.commit()
    except:
        db.rollback()
        raise

```

(In Python, the `@contextmanager` will cause the `yield` expression `raise` if the associated `with` exits with an exception.)

One may also note that we have two `yield` expressionss in the above example, depending on the state of the parameters. Remember that the requirement of `@contextmanager` is that a single `yield` expression evaluates once, but not that there only be one `yield` expression in the coroutine body. This is an important facet of `@contextmanager` that opens up a wealth of possibilities:

```

@contextmanager
def as_local_file(url: URL):
    'Get a locally-readable copy of the file at the given URL'
    if url.scheme == 'file':
        # The file is already a local file, so just yield that path
        yield Path(url.path)
    elif url.scheme == 'ssh':
        # Mount the parent directory using SSH
        with ssh_mount(url.with_parent_path()) as mount_dir:
            # Yield the path to the file
            yield mount_dir / url.path.filename

```

```

# The SSH host will be unmounted on scope exit
else:
    # Create a temporary dir that is deleted when we leave scope
    with temp_dir() as td:
        tmp_file: Path = td / 'tmp.bin'
        download_file_to(tmp_file, url)
        yield tmp_file

```

Now we have multiple completely different setup and cleanup behaviors depending on the scheme of url. Could you implement this with C++? One could imagine:

```

1 struct as_local_file {
2     variant<local_file_state,
3             ssh_mount_state,
4             download_tmpdir_state> _state;
5
6     as_local_file(url u); // Initialize the correct state
7
8     fs::path local_path(); // Get the path out of _state
9 };

```

(The implementation of the various states is left as an exercise to the reader...or you can read on...) Wouldn't it be simpler if we could just do the same thing as in Python?

```

1 co_resource<fs::path>
2 as_local_file(url u) {
3     if (u.scheme == "file") {
4         co_yield fs::path(u.path);
5     } else if (u.scheme == "ssh") {
6         auto mnt = open_ssh_mount(u.parent_path_url());
7         co_yield mnt.root / fs::path(u.path).filename();
8     } else {
9         auto tmpdir = make_tempdir();
10        auto tmp = tmpdir.path / "tmp.bin";
11        download_into(tmp, u);
12        co_yield tmp;
13    }
14 }
15
16 void print_content(url u) {
17     co_resource<fs::path> local = as_local_file(u);
18     fmt::print(read_file(*local));
19 }

```

In this example, `co_resource<T>` is a class template that implements the same machinery as `extmanager` from Python.

Not only is `co_resource<T>` entirely implementable: It is remarkably simple!

37.2 Implementing `co_resource`

A `co_resource` is the return type of a coroutine function that:

1. Runs the coroutine until the first `co_yield` expression.
2. Captures the operand of `co_yield` and suspends the coroutine.
3. The user of the resource can access the captured value using `operator*` and `operator->`.
4. When the `co_resource` is destroyed, the coroutine is resumed and expected to return without awaiting or yielding again.
5. The coroutine is destroyed.

37.2.1 Getting Started with Coroutines

C++20 coroutines are intimidating at first: There seems to be a lot going on compared to other language's coroutines. In Python all I have to do is call `yield` and my function suddenly becomes a coroutine! In C++, it is not so easy.

In my opinion, here's a very important preqrequisite to begin understanding C++ coroutines:

You should mentally detach the relationship between the return value and the body of the coroutine function: Including the return type itself!

Imagine a coroutine function:

```
coro_widget
my_lib::do_something(int arg) {
    // Code block containing 'co_await', 'co_yield', and/or 'co_return'
}
```

Traditionally, one would perceive `my_lib::do_something` as “a function that takes an `int` and returns a `coro_widget`”. But with coroutines, this is not the full picture. Instead, I recommend approaching with this mental model:

`my_lib::do_something` is a coroutine factory that accepts an `int` and generates a coroutine, and then returns a `coro_widget` with a handle to that coroutine.

Importantly: The block of code associated with `do_something` is not necessarily relevant in the creation of the returned `coro_widget` object! The types of the operands of `co_await`, `co_yield`, and `co_return` are orthogonal to the actual return value of `coro_widget`. Imagine it more like this:

```
1 struct __do_something {
2     using return_type = coro_widget;
3
4     // ... state data ...
5 }
```

```

6     void __impl() {
7         // Code block containing 'co_await', 'co_yield', and/or 'co_return'
8     }
9 };
10
11 coro_widget my_lib::do_something(int arg) {
12     auto co = new __co_magic<__do_something>{arg};
13     return co->__create_return_value();
14 }
```

This omits a ton of important details and is partially psuedo-code (the body of `__do_something`:`__impl` requires transforming the original coroutine function body), but is a closer model of what actually happens. We'll fill in some of the blanks below.

37.2.2 The Return Type and the Promise Type

When the coroutine object is being constructed, there are several customization points attached. The way these customizations are injected is via the declared return type and parameter types of the coroutine function. In particular, the compiler asks for the “promise type” via the `std::coroutine_traits` template:

```
using __promise_type =
    std::coroutine_traits<
        declared_return_type,
        param_types...>::promise_type;
```

One can specialize `std::coroutine_traits` arbitrarily, or rely on the default definition, which simply asks for `promise_type` on the declared return type itself. This gives us a simple beginning for `co_resource`:

```
template <typename T>
class co_resource {
public:
    struct promise_type {};
```

from here-on-out, the `promise_type` is fully in control of the coroutine, and the actual return type and parameter types of the initial coroutine function are not directly consulted again. Everything we want to do to control the coroutine must be exposed via the `promise_type`.

37.2.3 The Promise Object and the Coroutine State

The coroutine’s promise object and the opaque coroutine itself are intrinsically linked together. The coroutine can be accessed via `coroutine_handle<promise_type>`, and the promise can be accessed via the `promise()` method of the coroutine handle. The promise object will live exactly as long as the coroutine itself, and will not be moved or copied. To obtain the coroutine handle, one simply calls the static named-constructor `coroutine_handle<P>::from_promise(P& p)` with `P` as the promise type and `p` being the generated promise object. Thus the promise object can get a handle to its associated coroutine at any time:

```
void SomePromise::do_something() {
    std::coroutine_handle<SomePromise> my_coro =
        std::coroutine_handle<SomePromise>::from_promise(*this);
}
```

likewise, one can obtain a reference to the associated promise object with the `.promise()` method of `coroutine_handle`:

```
void f(coroutine_handle<SomePromise> coro) {
    SomePromise& p = coro.promise();
}
```

One should think of `coroutine_handle` as a reference to an opaque coroutine object. For `coroutine_handle<P>`, the coroutine's promise object is of type P. A `coroutine_handle<void>` (a.k.a. `coroutine_handle<>`) is a reference to a coroutine with an unknown promise type (and therefore the `.promise()` method is not available).

Because a `coroutine_handle<P>` and the promise object P can be easily obtained from one another, it helps to think of them as interchangeable objects that provide interfaces to different aspects of the underlying coroutine. So if your question is “should I use the `coroutine_handle<P>` or `P&` here?” the answer is “either one,” but storing the `coroutine_handle<P>` will often make the code more concise as it is a trivial type and it is more terse in code to obtain the promise via the handle than to obtain the handle via the promise.

The promise object is also the bridge between the coroutine and its caller. Any state that we wish to be accessible outside of the coroutine body should be declared as data in the promise type.

37.2.4 Some Boilerplate

Any promise object needs to provide a few methods that control the setup/teardown of the coroutine. For our case, these are not particularly interesting, so we can use some simple defaults:

```
1 struct promise_type {
2     auto initial_suspend() noexcept { return std::suspend_never{}; }
3     auto final_suspend() noexcept { return std::suspend_always{}; }
4     void return_void() noexcept {}
5     void unhandled_exception() { throw; }
6 };
```

For the above:

1. On `initial_suspend()`, we tell the compiler that the coroutine should begin execution immediately before returning to the caller.
2. On `final_suspend()`, we do nothing interesting and always suspend the coroutine when exiting.
3. `return_void` is called in case of `co_return`; or reaching the end of the coroutine function body. This is a no-op.
4. If an unhandled exception occurs, immediately re-`throw` it. This will cause the exception to throw out of the most recent call to `resume()` on the coroutine (including the implicit one inserted by the compiler that may occur following `initial_suspend()`).

37.2.5 Yielding a Value

We need to tell the coroutine machinery what to do when the coroutine evaluates a `co_yield` expression. This, unsurprisingly, is controlled via the promise object.

When a `co_yield` expression is evaluated, the operand is passed to the `yield_value()` method of the promise object. This gives us a chance to send that information back to the outside world. The return value of `yield_value()` is then given to a synthesized `co_await` expression, which determines whether or not to suspend the coroutine as part of that `co_yield`. In our case, we always want to suspend following the yield, so we return `suspend_always` again.

Because we are always suspending the coroutine after the yield, and the operand value has a lifetime of the full `co_yield` expression, the operand will live until the coroutine is resumed, meaning it is safe to store the address of the operand directly without making any moves or copies. Here’s what our `promise_type` looks like with these changes:

```

1 struct promise_type {
2     // ...
3     const T* yielded_value_ptr;
4
5     auto yield_value(const T& arg) noexcept {
6         yielded_value_ptr = &arg;
7         return std::suspend_always{};
8     }
9 };

```

We’re almost done. Now we just need to open up the communication channel between the coroutine caller and the coroutine state. This is done using the return object of the coroutine.

37.2.6 The Return Object

After constructing the promise object, the next operation inserted by the compiler is the construction of the “return object.”

The return object is not the `co_return` value of the coroutine, but rather the coroutine caller’s interface to the coroutine itself. The declared return type of the coroutine function is the type of the return object. It is the return value of the coroutine factory’s machinery that started the coroutine in the first place.

The return object is obtained by calling the `get_return_object()` method on the promise object. The return type of this method must be convertible to the declared return type of the coroutine function. Note that the conversion takes place after the `initial_suspend` (and possible `initial_resume()`): This fact will be important.

Given that the `coroutine_handle` gives us access to all of the state, the simplest way to implement the `co_resource<T>` is to store the coroutine handle:

```

1 template <typename T>
2 class co_resource {
3     // ...
4 private:

```

```

5     // The coroutine we are managing
6     coroutine_handle<promise_type> _coro;
7 };

```

Importantly, we want to perform exception handling as part of the construction of the `co_resource`, but we need this to happen after the initial-suspend, but before the coroutine return object is given to the caller. We do this by taking advantage of the `get_return_object()` being converted to the actual return object by using a simple intermediate type:

```

1 template <typename T>
2 class co_resource {
3     // ...
4
5 public:
6     struct init {
7         std::coroutine_handle<co_resource::promise_type> coro;
8     };
9
10    // Implicit convert from the intermediate object
11    co_resource(init i)
12        : _coro{i.coro} {}
13
14    struct promise_type {
15        // ...
16
17        auto get_return_object() {
18            // Create the intermediate object
19            return init{coroutine_handle<promise_type>::from_promise(*this)};
20        }
21    };
22};

```

Now we have a `co_resource` type that can be constructed, but how do we store and access the managed resource? When do we finish the coroutine to clean up the resource?

Firstly, to access the managed object: with the address of the yielded value stored in the promise, and that promise being given to the `co_resource` via the `coroutine_handle`, we can access that yielded value from the outside:

```

1 template <typename T>
2 class co_resource {
3     // ...
4
5 public:
6     const T& operator*() const noexcept {
7         return *_coro.promise().yielded_value_ptr;

```

```

8     }
9
10    const T* operator->() const noexcept {
11        return _coro.promise().yielded_value_ptr;
12    }
13};

```

Finally, to clean up: as part of the destructor of `co_resource`, we do the final `resume()` of the coroutine to run the cleanup code:

```

1 template <typename T>
2 class co_resource {
3     // ...
4 public:
5     ~co_resource() {
6         // Resume the coroutine from the co_yield point:
7         _coro.resume();
8         // Clean up:
9         _coro.destroy();
10    }
11};

```

We now we are ready to use the `co_resource`:

```

1 co_resource<string> greeting() {
2     fmt::print("Enter");
3     co_yield "Hello!";
4     fmt::print("Exit");
5 }
6
7 void foo() {
8     co_resource<string> r = greeting();
9     cout << *r;
10}

```

Here' s how `foo()` executes:

1. Enter the setup for `greeting()`
2. Create a coroutine state and `co_resource<string>::promise_type pr` object. These are stored in dynamic allocated state by default, but this allocation can be elided by a Sufficiently Smart Compiler.
3. Do init `_rv = pr.get_return_object()`
4. Because we asked the coroutine not to suspend initially, the coroutine begins execution.
5. We hit the `co_yield` in `greeting()`.
6. We pass the operand of `co_yield` to `yield_value()`

7. `yield_value()` stores the address of the argument in a pointer member.
8. We tell the coroutine to immediately suspend and return to the resumer as part of the `co_yield`.
9. The `__rv` object is returned from `greeting()`, causing the conversion from `init` to `co_resource<string>`, giving management of the coroutine to the `co_resource<string>`, now out-of-the-hands of the compiler's machinery. It is important that this conversion happens after the initial suspend and the first `co_yield` is hit, because if there is an exception before the `co_yield`, we do not want to give ownership of the coroutine to a `co_resource`, which would attempt to `resume()` the failed coroutine as part of its destructor.

If you run this code, you'll see "Enter" followed by "Hello!", then "Exit", and that's all there is to it.

37.3 What Else?

There's a few open questions with this simple implementation:

1. What about a `co_resource<void>`?
2. What about a `co_resource<T&>` (with a non-const reference)?
3. How can we assert that the coroutine yields exactly once?
4. If the user copies or moves the `co_resource` object, that will probably be very bad. What would move operations look like?
5. If the coroutine throws during the cleanup phase, this will `std::terminate()` since it occurs in the `noexcept` context of `~co_resource()`. Is this the best option? Why or why not?

I'll leave these as an exercise for the reader. You can see and use the fully-finished implementation from `neo-fun` here².

²https://github.com/vector-of-bool/neo-fun/blob/develop/src/neo/co_resource.hpp

Chapter 38

C++ Coroutines: Understanding the Compiler Transform

• Lewis Baker  2022-08-27  ★★★★☆

38.1 Introduction

Previous blogs in the series on “Understanding C++ Coroutines” talked about the different kinds of transforms the compiler performs on a coroutine and its `co_await`, `co_yield` and `co_return` expressions. These posts described how each expression was lowered by the compiler to calls to various customisation points/methods on user-defined types.

- Coroutine Theory¹
- C++ Coroutines: Understanding operator `co_await`²
- C++ Coroutines: Understanding the promise type³
- C++ Coroutines: Understanding Symmetric Transfer⁴

However, there was one part of these descriptions that may have left you unsatisfied. The all hand-waved over the concept of a “suspend-point” and said something vague like “the coroutine suspends here” and “the coroutine resumes here” but didn’t really go into detail about what that actually means or how it might be implemented by the compiler.

In this post I am going to go a bit deeper to show how all the concepts from the previous posts come together. I’ll show what happens when a coroutine reaches a suspend-point by walking through the lowering of a coroutine into equivalent non-coroutine, imperative C++ code.

Note that I am not going to describe exactly how a particular compiler lowers coroutines into machine code (compilers have extra tricks up their sleeves here), but rather just one possible lowering of coroutines into portable C++ code.

Warning: This is going to be a fairly deep dive!

¹<https://lewissbaker.github.io/2017/09/25/coroutine-theory>

²<https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await>

³<https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type>

⁴https://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer

38.2 Setting the Scene

For starters, let's assume we have a basic `task` type that acts as both an awaitable and a coroutine return-type. For the sake of simplicity, let's assume that this coroutine type allows producing a result of type `int` asynchronously.

In this post we are going to walk through how to lower the following coroutine function into C++ code that does not contain any of the coroutine keywords `co_await`, `co_return` so that we can better understand what this means.

```

1 // Forward declaration of some other function. Its implementation is not relevant.
2 task f(int x);
3
4 // A simple coroutine that we are going to translate to non-C++ code
5 task g(int x) {
6     int fx = co_await f(x);
7     co_return fx * fx;
8 }
```

38.3 Defining the task type

To begin, let us first declare the `task` class that we will be working with.

For the purposes of understanding how the coroutine is lowered, we do not need to know the definitions of the methods for this type. The lowering will just be inserting calls to them.

The definitions of these methods are not complicated, and I will leave them as an exercise for the reader as practice for understanding the previous posts.

```

1 class task {
2 public:
3     structawaiter;
4
5     class promise_type {
6     public:
7         promise_type() noexcept;
8         ~promise_type();
9
10        struct final_awaiter {
11            bool await_ready() noexcept;
12            std::coroutine_handle<> await_suspend(
13                std::coroutine_handle<promise_type> h) noexcept;
14            void await_resume() noexcept;
15        };
16
17        task get_return_object() noexcept;
18        std::suspend_always initial_suspend() noexcept;
```

```

19     final_awaiter final_suspend() noexcept;
20     void unhandled_exception() noexcept;
21     void return_value(int result) noexcept;
22
23 private:
24     friend task::awaiter;
25     std::coroutine_handle<> continuation_;
26     std::variant<std::monostate, int, std::exception_ptr> result_;
27 };
28
29 task(task&& t) noexcept;
30 ~task();
31 task& operator=(task&& t) noexcept;
32
33 struct awaiter {
34     explicit awaiter(std::coroutine_handle<promise_type> h) noexcept;
35     bool await_ready() noexcept;
36     std::coroutine_handle<promise_type> await_suspend(
37         std::coroutine_handle<> h) noexcept;
38     int await_resume();
39 private:
40     std::coroutine_handle<promise_type> coro_;
41 };
42
43 awaiter operator co_await() && noexcept;
44
45 private:
46     explicit task(std::coroutine_handle<promise_type> h) noexcept;
47
48     std::coroutine_handle<promise_type> coro_;
49 };

```

The structure of this task type should be familiar to those that have read the C++ Coroutines: Understanding Symmetric Transfer post⁵.

38.4 Step 1: Determining the promise type

```

1 task g(int x) {
2     int fx = co_await f(x);
3     co_return fx * fx;
4 }

```

⁵https://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer

When the compiler sees that this function contains one of the three coroutine keywords (`co_await`, `co_yield` or `co_return`) it starts the coroutine transformation process.

The first step here is determining the `promise_type` to use for this coroutine.

This is determined by substituting the return-type and argument-types of the signature as template arguments to the `std::coroutine_traits` type.

e.g. For our function, `g`, which has return type `task` and a single argument of type `int`, the compiler will look this up using `std::coroutine_traits<task, int>::promise_type`.

Let's define an alias so we can refer to this type later:

```
using __g_promise_t = std::coroutine_traits<task, int>::promise_type;
```

Note: I am using leading double-underscore here to indicate symbols internal to the compiler that the compiler generates. Such symbols are reserved by the implementation and should not be used in your own code.

Now, as we have not specialised `std::coroutine_traits` this will instantiate the primary template which just defines the nested `promise_type` as an alias of the nested `promise_type` name of the return-type. i.e. this should resolve to the type `task::promise_type` in our case.

38.5 Step 2: Creating the coroutine state

A coroutine function needs to preserve the state of the coroutine, parameters and local variables when it suspends so that they remain available when the coroutine is later resumed.

This state, in C++ standardese, is called the *coroutine state* and is typically heap allocated.

Let's start by defining a struct for the coroutine-state for the coroutine, `g`.

We don't know what the contents of this type are going to be yet, so let's just leave it empty for now.

```
struct __g_state {
    // to be filled out
};
```

The coroutine state contains a number of different things:

- The promise object
- Copies of any function parameters
- Information about the suspend-point that the coroutine is currently suspended at and how to resume/destroy it
- Storage for any local variables / temporaries whose lifetimes span a suspend-point

Let's start by adding storage for the promise object and parameter copies.

```
struct __g_state {
    int x;
    __g_promise_t __promise;

    // to be filled out
};
```

Next we should add a constructor to initialise these data-members.

Recall that the compiler will first attempt to call the promise constructor with lvalue-references to the parameter copies, if that call is valid, otherwise fall back to calling the default constructor of the promise type.

Let's create a simple helper to assist with this:

```

1 template<typename Promise, typename... Params>
2 Promise construct_promise([[maybe_unused]] Params&... params) {
3     if constexpr (std::constructible_from<Promise, Params&...>) {
4         return Promise(params...);
5     } else {
6         return Promise();
7     }
8 }
```

Thus the coroutine-state constructor might look something like this:

```

1 struct __g_state {
2     __g_state(int&& x)
3     : x(static_cast<int&&>(x))
4     , __promise(construct_promise<__g.promise_t>(x))
5     {}
6
7     int x;
8     __g.promise_t __promise;
9     // to be filled out
10};
```

Now that we have the beginnings of a type to represent the coroutine-state, let's also start to stub out the beginnings of the lowered implementation of `g()` by having it heap-allocate an instance of the `__g_state` type, passing the function parameters so they can be copied/ moved into the coroutine-state.

Some terminology - I use the term “ramp function” to refer to the part of the coroutine implementation containing the logic that initialises the coroutine state and gets it ready to start executing the coroutine. i.e. it is like an on-ramp for entering execution of the coroutine body.

```

task g(int x) {
    auto* state = new __g_state(static_cast<int&&>(x));
    // ... implement rest of the ramp function
}
```

Note that our promise-type does not define its own custom `operator new` overloads, and so we are just calling global `:operator new` here.

If the promise type did define a custom `operator new` then we'd call that instead of the global `:operator new`. We would first check whether `operator new` was callable with the argument list (size, paramLvalues...) and if so call it with that argument list. Otherwise, we'd call it with just the (size) argument list. The ability for the `operator new` to get access to the parameter list of the coroutine function is

sometimes called “parameter preview” and is useful in cases where you want to use an allocator passed as a parameter to allocate storage for the coroutine-state.

If the compiler found any definition of `__g_promise_t::operator new` then we’d lower to the following logic instead:

```

1 template<typename Promise, typename... Args>
2 void* __promise_allocate(std::size_t size, [[maybe_unused]] Args&... args) {
3     if constexpr (requires { Promise::operator new(size, args...); }) {
4         return Promise::operator new(size, args...);
5     } else {
6         return Promise::operator new(size);
7     }
8 }
9
10 task g(int x) {
11     void* state_mem = __promise_allocate<__g_promise_t>(sizeof(__g_state), x);
12     __g_state* state;
13     try {
14         state = ::new (state_mem) __g_state(static_cast<int&&>(x));
15     } catch (...) {
16         __g_promise_t::operator delete(state_mem);
17         throw;
18     }
19     // ... implement rest of the ramp function
20 }
```

Also, this promise-type does not define the `get_return_object_on_allocation_failure()` static member function. If this function is defined on the promise-type then the allocation here would instead use the `std::nothrow_t` form of `operator new` and upon returning `nullptr` would then `return __g_promise_t::get_return_object_on_allocation_failure();`.

i.e. it would look something like this instead:

```

1 task g(int x) {
2     auto* state = ::new (std::nothrow) __g_state(static_cast<int&&>(x));
3     if (state == nullptr) {
4         return __g_promise_t::get_return_object_on_allocation_failure();
5     }
6     // ... implement rest of the ramp function
7 }
```

For simplicity for the rest of the example, we’ll just use the simplest form that calls the global `::operator new` memory allocation function.

38.6 Step 3: Call `get_return_object()`

The next thing the ramp function does is to call the `get_return_object()` method on the promise object to obtain the return-value of the ramp function.

The return value is stored as a local variable and is returned at the end of the ramp function (after the other steps have been completed).

```

1 task g(int x) {
2     auto* state = new __g_state(static_cast<int&&>(x));
3     decltype(auto) return_value = state->_promise.get_return_object();
4     // ... implement rest of ramp function
5     return return_value;
6 }
```

However, now it's possible that the call to `get_return_object()` might throw, and in which case we want to free the allocated coroutine state. So for good measure, let's give ownership of the state to a `std::unique_ptr` so that it's freed in case a subsequent operation throws an exception:

```

1 task g(int x) {
2     std::unique_ptr<__g_state> state(new __g_state(static_cast<int&&>(x)));
3     decltype(auto) return_value = state->_promise.get_return_object();
4     // ... implement rest of ramp function
5     return return_value;
6 }
```

38.7 Step 4: The initial-suspend point

The next thing the ramp function does after calling `get_return_object()` is to start executing the body of the coroutine, and the first thing to execute in the body of the coroutine is the initial suspend-point. i.e. we evaluate `co_await promise.initial_suspend()`.

Now, ideally we'd just treat the coroutine as initially suspended and then just implement the launching of the coroutine as a resumption of the initially suspended coroutine. However, the specification of the initial-suspend point has a few quirks with regards to how it handles exceptions and the lifetime of the coroutine state. This was a late tweak to the semantics of the initial-suspend point just before C++20 was released to fix some perceived issues here.

Within the evaluation of the initial-suspend-point, if an exception is thrown either from:

- the call to `initial_suspend()`,
- the call to `operator co_await()` on the returned awaitable (if one is defined),
- the call to `await_ready()` on the awainer, or
- the call to `await_suspend()` on the awainer

Then the exception propagates back to the caller of the ramp function and the coroutine state is automatically destroyed.

If an exception is thrown either from:

- the call to `await_resume()`,
- the destructor of the object returned from `operator co_await()` (if applicable), or
- the destructor of the object returned from `initial_suspend()`

Then this exception is caught by the coroutine body and `promise.unhandled_exception()` is called.

This means we need to be a bit careful how we handle transforming this part, as some parts will need to live in the ramp function and other parts in the coroutine body.

Also, since the objects returned from `initial_suspend()` and (optionally) `operator co_await()` will have lifetimes that span a suspend-point (they are created before the point at which the coroutine suspends and are destroyed after it resumes) the storage for those objects will need to be placed in the coroutine state.

In our particular case, the type returned from `initial_suspend()` is `std::suspend_always`, which happens to be an empty, trivially constructible type. However, logically we still need to store an instance of this type in the coroutine state, so we'll add storage for it anyway just to show how this works.

This object will only be constructed at the point that we call `initial_suspend()`, so we need to add a data-member of a certain type that that allows us to explicitly control its lifetime.

To support this, let's first define a helper class, `manual_lifetime` that is trivially constructible and trivially destructible but that lets us explicitly construct/destruct the value stored there when we need to.

```

1  template<typename T>
2  struct manual_lifetime {
3      manual_lifetime() noexcept = default;
4      ~manual_lifetime() = default;
5
6      // Not copyable/movable
7      manual_lifetime(const manual_lifetime&) = delete;
8      manual_lifetime(manual_lifetime&&) = delete;
9      manual_lifetime& operator=(const manual_lifetime&) = delete;
10     manual_lifetime& operator=(manual_lifetime&&) = delete;
11
12     template<typename Factory>
13         requires
14             std::invocable<Factory&> &&
15             std::same_as<std::invoke_result_t<Factory&>, T>
16         T& construct_from(Factory factory) noexcept(std::is_nothrow_invocable_v<Factory&>) {
17             return *::new (static_cast<void*>(&storage)) T(factory());
18         }
19
20         void destroy() noexcept(std::is_nothrow_destructible_v<T>) {
21             std::destroy_at(std::launder(reinterpret_cast<T*>(&storage)));
22         }
23
24         T& get() & noexcept {
25             return *std::launder(reinterpret_cast<T*>(&storage));

```

```

26     }
27
28 private:
29     alignas(T) std::byte storage[sizeof(T)];
30 };

```

Note that the `construct_from()` method is designed to take a lambda here rather than taking the constructor arguments. This allows us to make use of the guaranteed copy-elision when initialising a variable with the result of a function-call to construct the object in-place. If it were instead to take the constructor arguments then we'd end up calling an extra move-constructor unnecessarily.

Now we can declare a data-member for the temporary returned by `promise.initial_suspend()` using this `manual_lifetime` structure.

```

1 struct __g_state {
2     __g_state(int&& x);
3
4     int x;
5     __g.promise_t __promise;
6     manual_lifetime<std::suspend_always> __tmp1;
7     // to be filled out
8 };

```

The `std::suspend_always` type does not have an `operator co_await()` so we do not need to reserve storage for an extra temporary for the result of that call here.

Once we've constructed this object by calling `initial_suspend()`, we then need to call the trio of methods to implement the `co_await` expression: `await_ready()`, `await_suspend()` and `await_resume()`.

When invoking `await_suspend()` we need to pass it a handle to the current coroutine. For now we can just call `std::coroutine_handle<__g.promise_t>::from_promise()` and pass a reference to that promise. We'll look at the internals of what this does a little later.

Also, the result of the call to `.await_suspend(handle)` has type `void` and so we do not need to consider whether to resume this coroutine or another coroutine after calling `await_suspend()` like we do for the `bool` and `coroutine_handle`-returning flavours.

Finally, as all of the method invocations on the `std::suspend_always` awainer are declared `noexcept`, we don't need to worry about exceptions. If they were potentially throwing then we'd need to add extra code to make sure that the temporary `std::suspend_always` object was destroyed before the exception propagated out of the ramp function.

Once we get to the point where `await_suspend()` has returned successfully or where we are about to start executing the coroutine body we enter the phase where we no longer need to automatically destroy the coroutine-state if an exception is thrown. So we can call `release()` on the `std::unique_ptr` owning the coroutine state to prevent it from being destroyed when we return from the function.

So now we can implement the first part of the initial-suspend expression as follows:

```

1 task g(int x) {
2     std::unique_ptr<__g_state> state(new __g_state(static_cast<int&&>(x)));

```

```

3     decltype(auto) return_value = state->_promise.get_return_object();
4
5     state->_tmp1.construct_from([&]() -> decltype(auto) {
6         return state->_promise.initial_suspend();
7     });
8     if (!state->_tmp1.get().await_ready()) {
9         //
10        // ... suspend-coroutine here
11        //
12        state->_tmp1.get().await_suspend(
13            std::coroutine_handle<__g_promise_t>::from_promise(state->_promise));
14
15        state.release();
16
17        // fall through to return statement below.
18    } else {
19        // Coroutine did not suspend.
20
21        state.release();
22
23        //
24        // ... start executing the coroutine body
25        //
26    }
27    return __return_val;
28 }

```

The call to `await_resume()` and the destructor of `_tmp1` will appear in the coroutine body and so they do not appear in the ramp function.

We now have a (mostly) functional evaluation of the initial-suspend point, but we still have a couple of TODO' s in the code for this ramp function. To be able to resolve these we will first need to take a detour to look at the strategy for suspending a coroutine and later resuming it.

38.8 Step 5: Recording the suspend-point

When a coroutine suspends, it needs to make sure it resumes at the same point in the control flow that it suspended at.

It also needs to keep track of which objects with automatic-storage duration are alive at each suspend-point so that it knows what needs to be destroyed if the coroutine is destroyed instead of being resumed.

One way to implement this is to assign each suspend-point in the coroutine a unique number and then store this in an integer data-member of the coroutine state.

Then whenever a coroutine suspends, it writes the number of the suspend-point at which it is suspending to the coroutine state, and when it is resumed/destroyed we then inspect this integer to see which suspend

point it was suspended at.

Note that this is not the only way of storing the suspend-point in the coroutine state, however all 3 major compilers (MSVC, Clang, GCC) use this approach as the time this post was authored (c. 2022). Another potential solution is to use separate resume/destroy function-pointers for each suspend-point, although we will not be exploring this strategy in this post.

So let's extend our coroutine-state with an integer data-member to store the suspend-point index and initialise it to zero (we'll always use this as the value for the initial-suspend point).

```

1 struct __g_state {
2     __g_state(int&& x);
3
4     int x;
5     __g.promise_t __promise;
6     int __suspend_point = 0; // <-- add the suspend-point index
7     manual_lifetime<std::suspend_always> __tmp1;
8     // to be filled out
9 };

```

38.9 Step 6: Implementing `coroutine_handle::resume()` and `coroutine_handle::destroy()`

When a coroutine is resumed by calling `coroutine_handle::resume()` we need this to end up invoking some function that implements the rest of the body of the suspended coroutine. The invoked body function can then look up the suspend-point index and jump to the appropriate point in the control-flow.

We also need to implement the `coroutine_handle::destroy()` function so that it invokes the appropriate logic to destroy any in-scope objects at the current suspend-point and we need to implement `coroutine_handle::done()` to query whether the current suspend-point is a final-suspend-point.

The interface of the `coroutine_handle` methods does not know about the concrete coroutine state type - the `coroutine_handle<void>` type can point to any coroutine instance. This means we need to implement them in a way that type-erases the coroutine state type.

We can do this by storing function-pointers to the resume/destroy functions for that coroutine type and having `coroutine_handle::resume/destroy()` invoke those function-pointers.

The `coroutine_handle` type also needs to be able to be converted to/from a `void*` using the `coroutine_handle::address()` and `coroutine_handle::from_address()` methods.

Furthermore, the coroutine can be resumed/destroyed from any handle to that coroutine - not just the handle that was passed to the most recent `await_suspend()` call.

These requirements lead us to define the `coroutine_handle` type so that it only contains a pointer to the coroutine-state and that we store the resume/destroy function pointers as data-members of the coroutine state, rather than, say, storing the resume/destroy function pointers in the `coroutine_handle`.

Also, since we need the `coroutine_handle` to be able to point to an arbitrary coroutine-state object we need the layout of the function-pointer data-members to be consistent across all coroutine-state types.

One straight forward way of doing this is having each coroutine-state type inherit from some base-class that contains these data-members.

e.g. We can define the following type as the base-class for all coroutine-state types

```

1 struct __coroutine_state {
2     using __resume_fn = void(__coroutine_state*);
3     using __destroy_fn = void(__coroutine_state*);
4
5     __resume_fn* __resume;
6     __destroy_fn* __destroy;
7 };

```

Then the `coroutine_handle::resume()` method can simply call `__resume()`, passing a pointer to the `__coroutine_state` object. Similarly, we can do this for the `coroutine_handle::destroy()` method and the `__destroy` function-pointer.

For the `coroutine_handle::done()` method, we choose to treat a null `__resume` function pointer as an indication that we are at a final-suspend-point. This is convenient since the final suspend point does not support `resume()`, only `destroy()`. If someone tries to call `resume()` on a coroutine suspended at the final-suspend-point (which has undefined-behaviour) then they end up calling a null function pointer which should fail pretty quickly and point out their error.

Given this, we can implement the `coroutine_handle<void>` type as follows:

```

1 namespace std
2 {
3     template<typename Promise = void>
4     class coroutine_handle;
5
6     template<>
7     class coroutine_handle<void> {
8     public:
9         coroutine_handle() noexcept = default;
10        coroutine_handle(const coroutine_handle&) noexcept = default;
11        coroutine_handle& operator=(const coroutine_handle&) noexcept = default;
12
13        void* address() const {
14            return static_cast<void*>(state_);
15        }
16
17        static coroutine_handle from_address(void* ptr) {
18            coroutine_handle h;
19            h.state_ = static_cast<__coroutine_state*>(ptr);
20            return h;
21        }
22
23        explicit operator bool() noexcept {
24            return state_ != nullptr;
25        }

```

```

26
27     friend bool operator==(coroutine_handle a, coroutine_handle b) noexcept {
28         return a.state_ == b.state_;
29     }
30
31     void resume() const {
32         state_->__resume(state_);
33     }
34     void destroy() const {
35         state_->__destroy(state_);
36     }
37
38     bool done() const {
39         return state_->__resume == nullptr;
40     }
41
42 private:
43     __coroutine_state* state_ = nullptr;
44 };
45 }
```

38.10 Step 7: Implementing `coroutine_handle<Promise>::promise()` and `from_promise()`

For the more general `coroutine_handle<Promise>` specialisation, most of the implementations can just reuse the `coroutine_handle<void>` implementations. However, we also need to be able to get access to the promise object of the coroutine-state, returned from the `promise()` method, and also construct a `coroutine_handle` from a reference to the promise-object.

However, again we cannot simply point to the concrete coroutine state type since the `coroutine_handle<Promise>` type must be able to refer to any coroutine-state whose promise-type is `Promise`.

We need to define a new coroutine-state base-class that inherits from `__coroutine_state` and which contains the promise object so we can then define all coroutine-state types that use a particular promise-type to inherit from this base-class.

```

1 template<typename Promise>
2 struct __coroutine_state_with.promise : __coroutine_state {
3     __coroutine_state_with.promise() noexcept {}
4     ~__coroutine_state_with.promise() {}
5
6     union {
7         Promise __promise;
8     };
9 }
```

You might be wondering why we declare the `__promise` member inside an anonymous union here...

The reason for this is that the derived class created for a particular coroutine function contains the definition for the argument-copy data-members. Data members from derived classes are by default initialised after data-members of any base-classes, so declaring the promise object as a normal data-member would mean that the promise object was constructed before the argument-copy data-members.

However, we need the constructor of the promise to be called after the constructor of the argument-copies - references to the argument-copies might need to be passed to the promise constructor.

So we reserve storage for the promise object in this base-class so that it has a consistent offset from the start of the coroutine-state, but leave the derived class responsible for calling the constructor/destructor at the appropriate point after the argument-copies have been initialised. Declaring the `__promise` as a union-member provides this control.

Let's update the `__g_state` class to now inherit from this new base-class.

```

1  struct __g_state : __coroutine_state_with_promise<__g_promise_t> {
2
3      __g_state(int&& __x)
4      : x(static_cast<int&&>(__x)) {
5          // Use placement-new to initialise the promise object in the base-class
6          ::new ((void*)std::addressof(this->__promise))
7              __g_promise_t(construct_promise<__g_promise_t>(x));
8      }
9
10     ~__g_state() {
11         // Also need to manually call the promise destructor before the
12         // argument objects are destroyed.
13         this->__promise.~__g_promise_t();
14     }
15
16     int __suspend_point = 0;
17     int x;
18     manual_lifetime<std::suspend_always> __tmp1;
19     // to be filled out
20 };

```

Now that we have defined the promise-base-class we can now implement the `std::coroutine_handle<Promise>` class template.

Most of the implementation should be largely identical to the equivalent methods in `coroutine_handle<void>` except with a `__coroutine_state_with_promise<Promise>` pointer instead of `__coroutine_state` pointer.

The only new part is the addition of the `promise()` and `from_promise()` functions.

- The `promise()` method is straight-forward - it just returns a reference to the `__promise` member of the coroutine-state.
- The `from_promise()` method requires us to calculate the address of the coroutine-state from the address of the promise object. We can do this by just subtracting the offset of the `__promise` member

from the address of the promise object.

Implementation of `coroutine_handle<Promise>`:

```

1  namespace std
2  {
3      template<typename Promise>
4      class coroutine_handle {
5          using state_t = __coroutine_state_with_promise<Promise>;
6      public:
7          coroutine_handle() noexcept = default;
8          coroutine_handle(const coroutine_handle&) noexcept = default;
9          coroutine_handle& operator=(const coroutine_handle&) noexcept = default;
10
11         operator coroutine_handle<void>() const noexcept {
12             return coroutine_handle<void>::from_address(address());
13         }
14
15         explicit operator bool() const noexcept {
16             return state_ != nullptr;
17         }
18
19         friend bool operator==(coroutine_handle a, coroutine_handle b) noexcept {
20             return a.state_ == b.state_;
21         }
22
23         void* address() const {
24             return static_cast<void*>(static_cast<__coroutine_state*>(state_));
25         }
26
27         static coroutine_handle from_address(void* ptr) {
28             coroutine_handle h;
29             h.state_ = static_cast<state_t*>(static_cast<__coroutine_state*>(ptr));
30             return h;
31         }
32
33         Promise& promise() const {
34             return state_->__promise;
35         }
36
37         static coroutine_handle from_promise(Promise& promise) {
38             coroutine_handle h;
39
// We know the address of the __promise member, so calculate the

```

```

41         // address of the coroutine-state by subtracting the offset of
42         // the __promise field from this address.
43         h.state_ = reinterpret_cast<state_t*>(
44             reinterpret_cast<unsigned char*>(std::addressof(promise)) -
45             offsetof(state_t, __promise));
46
47         return h;
48     }
49
50     // Define these in terms of their `coroutine_handle<void>` implementations
51
52     void resume() const {
53         static_cast<coroutine_handle<void>>(*this).resume();
54     }
55
56     void destroy() const {
57         static_cast<coroutine_handle<void>>(*this).destroy();
58     }
59
60     bool done() const {
61         return static_cast<coroutine_handle<void>>(*this).done();
62     }
63
64     private:
65         state_t* state_;
66     };
67 }
```

Now that we have defined the mechanism by which coroutines are resumed, we can now return to our “ramp” function and update it to initialise the new function-pointer data-members we’ve added to the coroutine-state.

38.11 Step 8: The beginnings of the coroutine body

Let’s now forward-declare resume/destroy functions of the right signature and update the `__g_state` constructor to initialise the coroutine-state so that the resume/destroy function-pointers point at them:

```

1 void __g_resume(__coroutine_state* s);
2 void __g_destroy(__coroutine_state* s);
3
4 struct __g_state : __coroutine_state_with_promise<__g.promise_t> {
5     __g_state(int&& __x)
6     : x(static_cast<int&&>(__x)) {
7         // Initialise the function-pointers used by coroutine_handle methods.
```

```

8     this->__resume = &__g_resume;
9     this->__destroy = &__g_destroy;
10
11     // Use placement-new to initialise the promise object in the base-class
12     ::new ((void*)std::addressof(this->__promise))
13         __g.promise_t(construct_promise<__g.promise_t>(x));
14     }
15
16     // ... rest omitted for brevity
17 };
18
19
20 task g(int x) {
21     std::unique_ptr<__g_state> state(new __g_state(static_cast<int&&>(x)));
22     decltype(auto) return_value = state->__promise.get_return_object();
23
24     state->__tmp1.construct_from([&]() -> decltype(auto)) {
25         return state->__promise.initial_suspend();
26     });
27     if (!state->__tmp1.get().await_ready()) {
28         state->__tmp1.get().await_suspend(
29             std::coroutine_handle<__g.promise_t>::from_promised(state->__promise));
30         state.release();
31         // fall through to return statement below.
32     } else {
33         // Coroutine did not suspend. Start executing the body immediately.
34         __g_resume(state.release());
35     }
36     return return_value;
37 }
```

This now completes the ramp function and we can now focus on the resume/destroy functions for g().

Let's start by completing the lowering of the initial-suspend expression.

When __g_resume() is called and the __suspend_point index is 0 then we need it to resume by calling await_resume() on __tmp1 and then calling the destructor of __tmp1.

```

1 void __g_resume(__coroutine_state* s) {
2     // We know that 's' points to a __g_state.
3     auto* state = static_cast<__g_state*>(s);
4
5     // Generate a jump-table to jump to the correct place in the code based
6     // on the value of the suspend-point index.
7     switch (state->__suspend_point) {
8         case 0: goto suspend_point_0;
```

```

9     default: std::unreachable();
10    }
11
12  suspend_point_0:
13      state->__tmp1.get().await_resume();
14      state->__tmp1.destroy();
15
16      // TODO: Implement rest of coroutine body.
17      //
18      //  int fx = co_await f(x);
19      //  co_return fx * fx;
20  }
```

And when `__g_destroy()` is called and the `__suspend_point` index is 0 then we need it to just destroy `__tmp1` before then destroying and freeing the coroutine-state.

```

1 void __g_destroy(__coroutine_state* s) {
2     auto* state = static_cast<__g_state*>(s);
3
4     switch (state->__suspend_point) {
5         case 0: goto suspend_point_0;
6         default: std::unreachable();
7     }
8
9  suspend_point_0:
10     state->__tmp1.destroy();
11     goto destroy_state;
12
13     // TODO: Add extra logic for other suspend-points here.
14
15 destroy_state:
16     delete state;
17 }
```

38.12 Step 9: Lowering the `co_await` expression

Next, let's take a look at lowering the `co_await f(x)` expression.

First we need to evaluate `f(x)` which returns a temporary task object.

As the temporary task is not destroyed until the semicolon at the end of the statement and the statement contains a `co_await` expression, the lifetime of the task therefore spans a suspend-point and so it must be stored in the coroutine-state.

When the `co_await` expression is then evaluated on this temporary task, we need to call the `operator co_await()` method which returns a temporary awaiter object. The lifetime of this object also spans the suspend-point and so must be stored in the coroutine-state.

Let's add the necessary members to the `__g_state` type:

```

1 struct __g_state : __coroutine_state_with_promise<__g_promise_t> {
2     __g_state(int&& __x);
3     ~__g_state();
4
5     int __suspend_point = 0;
6     int x;
7     manual_lifetime<std::suspend_always> __tmp1;
8     manual_lifetime<task> __tmp2;
9     manual_lifetime<task::awaiter> __tmp3;
10 };

```

Then we can update the `__g_resume()` function to initialise these temporaries and then evaluate the `3 await_ready, await_suspend` and `await_resume` calls that comprise the rest of the `co_await` expression.

Note that the `task::awaiter::await_suspend()` method returns a coroutine-handle so we need to generate code that resumes the returned handle.

We also need to update the suspend-point index before calling `await_suspend()` (we'll use the index 1 for this suspend-point) and then add an extra entry to the jump-table to ensure that we resume back at the right spot.

```

1 void __g_resume(__coroutine_state* s) {
2     // We know that 's' points to a __g_state.
3     auto* state = static_cast<__g_state*>(s);
4
5     // Generate a jump-table to jump to the correct place in the code based
6     // on the value of the suspend-point index.
7     switch (state->__suspend_point) {
8         case 0: goto suspend_point_0;
9         case 1: goto suspend_point_1; // <-- add new jump-table entry
10        default: std::unreachable();
11    }
12
13    suspend_point_0:
14        state->__tmp1.get().await_resume();
15        state->__tmp1.destroy();
16
17        // int fx = co_await f(x);
18        state->__tmp2.construct_from([&] {
19            return f(state->x);
20        });
21        state->__tmp3.construct_from([&] {
22            return static_cast<task&&>(state->__tmp2.get()).operator co_await();

```

```

23     });
24     if (!state->_tmp3.get().await_ready()) {
25         // mark the suspend-point
26         state->_suspend_point = 1;
27
28         auto h = state->_tmp3.get().await_suspend(
29             std::coroutine_handle<__g_promise_t>::from_promise(state->_promise));
30
31         // Resume the returned coroutine-handle before returning.
32         h.resume();
33
34         return;
35     }
36
37     suspend_point_1:
38         int fx = state->_tmp3.get().await_resume();
39         state->_tmp3.destroy();
40         state->_tmp2.destroy();
41
42         // TODO: Implement
43         // co_return fx * fx;
44     }

```

Note that the `int fx` local variable has a lifetime that does not span a suspend-point and so it does not need to be stored in the coroutine-state. We can just store it as a normal local variable in the `_g_resume` function.

We also need to add the necessary entry to the `_g_destroy()` function to handle when the coroutine is destroyed at this suspend-point.

```

1 void _g_destroy(__coroutine_state* s) {
2     auto* state = static_cast<__g_state*>(s);
3
4     switch (state->_suspend_point) {
5         case 0: goto suspend_point_0;
6         case 1: goto suspend_point_1; // <-- add new jump-table entry
7         default: std::unreachable();
8     }
9
10    suspend_point_0:
11        state->_tmp1.destroy();
12        goto destroy_state;
13
14    suspend_point_1:
15        state->_tmp3.destroy();
16        state->_tmp2.destroy();

```

```

17     goto destroy_state;
18
19     // TODO: Add extra logic for other suspend-points here.
20
21 destroy_state:
22     delete state;
23 }
```

So now we have finished implementing the statement:

```
int fx = co_await f(x);
```

However, the function `f(x)` is not marked `noexcept` and so it can potentially throw an exception. Also, the `awaiter::await_resume()` method is also not marked `noexcept` and can also potentially throw an exception.

When an exception is thrown from a coroutine-body the compiler generates code to catch the exception and then invoke `promise.unhandled_exception()` to give the promise an opportunity to do something with the exception. Let's look at implementing this aspect next.

38.13 Step 10: Implementing `unhandled_exception()`

The specification for coroutine definitions [dcl.fct.def.coroutine]⁶ says that the coroutine behaves as if its function-body were replaced by:

```

1  {
2      promise-type promise promise-constructor-arguments ;
3      try {
4          co_await promise.initial_suspend() ;
5          function-body
6      } catch ( ... ) {
7          if (!initial-await-resume-called)
8              throw ;
9          promise.unhandled_exception() ;
10     }
11 final-suspend :
12     co_await promise.final_suspend() ;
13 }
```

We have already handled the `initial-await_resume-called` branch separately in the `ramp` function, so we don't need to worry about that here.

Let's adjust the `_g_resume()` function to insert the try/catch block around the body.

Note that we need to be careful to put the `switch` that jumps to the right place inside the try-block as we are not allowed to enter a try-block using a `goto`.

⁶<https://eel.is/c++draft/dcl.fct.def.coroutine>

Also, we need to be careful to call `.resume()` on the coroutine handle returned from `await_suspend()` outside of the try/catch block. If an exception is thrown from the call `.resume()` on the returned coroutine then it should not be caught by the current coroutine, but should instead propagate out of the call to `resume()` that resumed this coroutine. So we stash the coroutine-handle in a variable declared at the top of the function and then `goto` a point outside of the try/catch and execute the call to `.resume()` there.

```

1 void __g_resume(__coroutine_state* s) {
2     auto* state = static_cast<__g_state*>(s);
3
4     std::coroutine_handle<void> coro_to_resume;
5
6     try {
7         switch (state->__suspend_point) {
8             case 0: goto suspend_point_0;
9             case 1: goto suspend_point_1; // <-- add new jump-table entry
10            default: std::unreachable();
11        }
12
13    suspend_point_0:
14        state->__tmp1.get().await_resume();
15        state->__tmp1.destroy();
16
17        // int fx = co_await f(x);
18        state->__tmp2.construct_from([&] {
19            return f(state->x);
20        });
21        state->__tmp3.construct_from([&] {
22            return static_cast<task&&>(state->__tmp2.get()).operator co_await();
23        });
24
25        if (!state->__tmp3.get().await_ready()) {
26            state->__suspend_point = 1;
27            coro_to_resume = state->__tmp3.get().await_suspend(
28                std::coroutine_handle<__g_promise_t>::from_promise(state->__promise));
29            goto resume_coro;
30        }
31
32    suspend_point_1:
33        int fx = state->__tmp3.get().await_resume();
34        state->__tmp3.destroy();
35        state->__tmp2.destroy();
36
37        // TODO: Implement

```

```

38     // co_return fx * fx;
39 } catch (...) {
40     state->_promise.unhandled_exception();
41     goto final_suspend;
42 }
43
44 final_suspend:
45     // TODO: Implement
46     // co_await promise.final_suspend();
47
48 resume_coro:
49     coro_to_resume.resume();
50     return;
51 }
```

There is a bug in the above code, however. In the case that the `__tmp3.get().await_resume()` call exits with an exception, we would fail to call the destructors of `__tmp3` and `__tmp2` before catching the exception.

Note that we cannot simply catch the exception, call the destructors and rethrow the exception here as this would change the behaviour of those destructors if they were to call `std::unhandled_exceptions()` since the exception would be “handled”. However if the destructor calls this during exception unwind, then call to `std::unhandled_exceptions()` should return non-zero.

We can instead define an RAII helper class to ensure that the destructors get called on scope exit in the case an exception is thrown.

```

1 template<typename T>
2 struct destructor_guard {
3     explicit destructor_guard(manual_lifetime<T>& obj) noexcept
4         : ptr_(std::addressof(obj))
5     {}
6
7     // non-movable
8     destructor_guard(destructor_guard&&) = delete;
9     destructor_guard& operator=(destructor_guard&&) = delete;
10
11 ~destructor_guard() noexcept(std::is_nothrow_destructible_v<T>) {
12     if (ptr_ != nullptr) {
13         ptr_->destroy();
14     }
15 }
16
17 void cancel() noexcept { ptr_ = nullptr; }
18
19 private:
```

```

20     manual_lifetime<T>* ptr_;
21 };
22
23 // Partial specialisation for types that don't need their destructors called.
24 template<typename T>
25     requires std::is_trivially_destructible_v<T>
26 struct destructor_guard<T> {
27     explicit destructor_guard(manual_lifetime<T>&) noexcept {}
28     void cancel() noexcept {}
29 };
30
31 // Class-template argument deduction to simplify usage
32 template<typename T>
33 destructor_guard(manual_lifetime<T>& obj) -> destructor_guard<T>;

```

Using this utility, we can now use this type to ensure that variables stored in the coroutine-state are destroyed when an exception is thrown.

Let's also use this class to call the destructors of the existing variables so that it also calls their destructors when they naturally go out of scope.

```

1 void __g_resume(__coroutine_state* s) {
2     auto* state = static_cast<__g_state*>(s);
3
4     std::coroutine_handle<void> coro_to_resume;
5
6     try {
7         switch (state->__suspend_point) {
8             case 0: goto suspend_point_0;
9             case 1: goto suspend_point_1; // <-- add new jump-table entry
10            default: std::unreachable();
11        }
12
13    suspend_point_0:
14        {
15            destructor_guard tmp1_dtor{state->__tmp1};
16            state->__tmp1.get().await_resume();
17        }
18
19        // int fx = co_await f(x);
20        {
21            state->__tmp2.construct_from([&] {
22                return f(state->x);
23            });
24            destructor_guard tmp2_dtor{state->__tmp2};

```

```

25
26     state->__tmp3.construct_from([&] {
27         return static_cast<task&&>(state->__tmp2.get()).operator co_await();
28     });
29     destructor_guard tmp3_dtor{state->__tmp3};
30
31     if (!state->__tmp3.get().await_ready()) {
32         state->__suspend_point = 1;
33
34         coro_to_resume = state->__tmp3.get().await_suspend(
35             std::coroutine_handle<__g_promise_t>::from_promise(state->__promise));
36
37         // A coroutine suspends without exiting scopes.
38         // So cancel the destructor-guards.
39         tmp3_dtor.cancel();
40         tmp2_dtor.cancel();
41
42         goto resume_coro;
43     }
44
45     // Don't exit the scope here.
46     //
47     // We can't 'goto' a label that enters the scope of a variable with a
48     // non-trivial destructor. So we have to exit the scope of the destructor
49     // guards here without calling the destructors and then recreate them after
50     // the `suspend_point_1` label.
51     tmp3_dtor.cancel();
52     tmp2_dtor.cancel();
53 }
54
55 suspend_point_1:
56     int fx = [&]() -> decltype(auto) {
57         destructor_guard tmp2_dtor{state->__tmp2};
58         destructor_guard tmp3_dtor{state->__tmp3};
59         return state->__tmp3.get().await_resume();
60     }();
61
62     // TODO: Implement
63     // co_return fx * fx;
64 } catch (...) {
65     state->__promise.unhandled_exception();
66     goto final_suspend;

```

```

67     }
68
69 final_suspend:
70     // TODO: Implement
71     // co_await promise.final_suspend();
72
73 resume_coro:
74     coro_to_resume.resume();
75     return;
76 }
```

Now our coroutine body will now destroy local variables correctly in the presence of any exceptions and will correctly call `promise.unhandled_exception()` if those exceptions propagate out of the coroutine body.

It's worth noting here that there can also be special handling needed for the case where the `promise.unhandled_exception()` method itself exits with an exception (e.g. if it rethrows the current exception).

In this case, the coroutine would need to catch the exception, mark the coroutine as suspended at a final-suspend-point, and then rethrow the exception.

For example: The `__g_resume()` function's catch-block would need to look like this:

```

1 try {
2     // ...
3 } catch (...) {
4     try {
5         state->__promise.unhandled_exception();
6     } catch (...) {
7         state->__suspend_point = 2;
8         state->__resume = nullptr; // mark as final-suspend-point
9         throw;
10    }
11 }
```

and we'd need to add an extra entry to the `__g_destroy` function's jump table:

```

1 switch (state->__suspend_point) {
2     case 0: goto suspend_point_0;
3     case 1: goto suspend_point_1;
4     case 2: goto destroy_state; // no variables in scope that need to be destroyed
// just destroy the coroutine-state object.
5
6 }
```

Note that in this case, the final-suspend-point is not necessarily the same suspend-point as the final-suspend-point as the `co_await promise.final_suspend()` suspend-point.

This is because the `promise.final_suspend()` suspend-point will often have some extra temporary objects related to the `co_await` expression which need to be destroyed when `coroutine_handle::de`

`stroy()` is called. Whereas, if `promise.unhandled_exception()` exits with an exception then those temporary objects will not exist and so won't need to be destroyed by `coroutine_handle::destroy()`.

38.14 Step 11: Implementing `co_return`

The next step is to implement the `co_return fx * fx;` statement.

This is relatively straight-forward compared to some of the previous steps.

The `co_return <expr>` statement gets mapped to:

```
promise.return_value(<expr>);
goto final-suspend-point;
```

So we can simply replace the TODO comment with:

```
state->__promise.return_value(fx * fx);
goto final_suspend;
```

Easy.

38.15 Step 12: Implementing `final_suspend()`

The final TODO in the code is now to implement the `co_await promise.final_suspend()` statement.

The `final_suspend()` method returns a temporary `task::promise_type::final_awaiter` type, which will need to be stored in the coroutine-state and destroyed in `__g_destroy`.

This type does not have its own `operator co_await()`, so we don't need an additional temporary object for the result of that call.

Like the `task::awaiter` type, this also uses the coroutine-handle-returning form of `await_suspend()`. So we need to ensure that we call `resume()` on the returned handle.

If the coroutine does not suspend at the final-suspend-point then the coroutine-state is implicitly destroyed. So we need to delete the state object if execution reaches the end of the coroutine.

Also, as all of the final-suspend logic is required to be noexcept, we don't need to worry about exceptions being thrown from any of the sub-expressions here.

Let's first add the data-member to the `__g_state` type.

```
1 struct __g_state : __coroutine_state_with_promise<__g.promise_t> {
2     __g_state(int&& __x);
3     ~__g_state();
4
5     int __suspend_point = 0;
6     int x;
7     manual_lifetime<std::suspend_always> __tmp1;
8     manual_lifetime<task> __tmp2;
9     manual_lifetime<task::awaiter> __tmp3;
10    manual_lifetime<task::promise_type::final_awaiter> __tmp4; // <---
11};
```

Then we can implement the body of the final-suspend expression as follows:

```

1  final_suspend:
2  // co_await promise.final_suspend
3  {
4      state->__tmp4.construct_from([&]() noexcept {
5          return state->__promise.final_suspend();
6      });
7      destructor_guard tmp4_dtor{state->__tmp4};
8
9      if (!state->__tmp4.get().await_ready()) {
10         state->__suspend_point = 2;
11         state->__resume = nullptr; // mark as final suspend-point
12
13         coro_to_resume = state->__tmp4.get().await_suspend(
14             std::coroutine_handle<__g_promise_t>::from_promise(state->__promise));
15
16         tmp4_dtor.cancel();
17         goto resume_coro;
18     }
19
20     state->__tmp4.get().await_resume();
21 }
22
23 // Destroy coroutine-state if execution flows off end of coroutine
24 delete state;
25 return;

```

And now we also need to update the `_g_destroy` function to handle this new suspend-point.

```

1 void _g_destroy(__coroutine_state* state) {
2     auto* state = static_cast<__g_state*>(s);
3
4     switch (state->__suspend_point) {
5         case 0: goto suspend_point_0;
6         case 1: goto suspend_point_1;
7         case 2: goto suspend_point_2;
8         default: std::unreachable();
9     }
10
11 suspend_point_0:
12     state->__tmp1.destroy();
13     goto destroy_state;
14

```

```

15 suspend_point_1:
16     state->__tmp3.destroy();
17     state->__tmp2.destroy();
18     goto destroy_state;
19
20 suspend_point_2:
21     state->__tmp4.destroy();
22     goto destroy_state;
23
24 destroy_state:
25     delete state;
26 }
```

We now have a fully functional lowering of the `g()` coroutine function.

We’re done! That’s it!

Or is it….

38.16 Step 13: Implementing symmetric-transfer and the noop-coroutine

It turns out there is actually a problem with the way we have implemented our `__g_resume()` function above.

The problems with this were discussed in more detail in the previous blog post so if you want to understand the problem more deeply please take a look at the post C++ Coroutines: Understanding Symmetric Transfer⁷.

The specification for `[expr.await]8` gives a little hint about how we should be handling the coroutine-handle-returning flavour of `await_suspend`:

If the type of await-suspend is `std::coroutine_handle<Z>`, await-suspend.`resume()` is evaluated.

[Note 1: This resumes the coroutine referred to by the result of await-suspend. Any number of coroutines can be successively resumed in this fashion, eventually returning control flow to the current coroutine caller or resumer ([dcl.fct.def.coroutine]⁹). — end note]

The note there, while non-normative and thus non-binding, is strongly encouraging compilers to implement this in such a way that it performs a tail-call to resume the next coroutine rather than resuming the next coroutine recursively. This is because resuming the next coroutine recursively can easily lead to unbounded stack growth if coroutines resume each other in a loop.

The problem is that we are calling `.resume()` on the next coroutine from within the body of the `__g_resume()` function and then returning, so the stack space used by the `__g_resume()` frame is not freed until after the next coroutine suspends and returns.

⁷https://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer

⁸<https://eel.is/c++draft/expr.await>

⁹<https://eel.is/c++draft/dcl.fct.def.coroutine>

Compilers are able to do this by implementing the resumption of the next coroutine as a tail-call. In this way, the compiler generates code that first pops the current stack frame, preserving the return-address, and then executes a `jmp` to the next coroutine's resume-function.

As we don't have a mechanism in C++ to specify that a function-call in the tail-position should be a tail-call we will need to instead actually return from the resume-function so that its stack-space can be freed, and then have the caller resume the next coroutine.

As the next coroutine may also need to resume another coroutine when it suspends, and this may happen indefinitely, the caller will need to resume the coroutines in a loop.

Such a loop is typically called a "trampoline loop" as we return back to the loop from one coroutine and then "bounce" off the loop back into the next coroutine.

If we modify the signature of the resume-function to return a pointer to the next coroutine's coroutine-state instead of returning void, then the `coroutine_handle::resume()` function can then just immediately call the `__resume()` function-pointer for the next coroutine to resume it.

Let's change the signature of the `__resume_fn` for a `__coroutine_state`:

```

1 struct __coroutine_state {
2     using __resume_fn = __coroutine_state* (__coroutine_state*);
3     using __destroy_fn = void (__coroutine_state*);
4
5     __resume_fn* __resume;
6     __destroy_fn* __destroy;
7 };

```

Then we can write the `coroutine_handle::resume()` function something like this:

```

1 void std::coroutine_handle<void>::resume() const {
2     __coroutine_state* s = state_;
3     do {
4         s = s->__resume(s);
5     } while /* some condition */;
6 }

```

The next question then becomes: "What should the condition be?"

This is where the `std::noop_coroutine()` helper comes into the picture.

The `std::noop_coroutine()` is a factory function that returns a special coroutine handle that has a no-op `resume()` and `destroy()` method. If a coroutine suspends and returns the noop-coroutine-handle from the `await_suspend()` method then this indicates that there is no more coroutine to resume and that the invocation of `coroutien_handle::resume()` that resumed this coroutine should return to its caller.

So we need to implement `std::noop_coroutine()` and the condition in `coroutine_handle::resume()` so that the condition returns false and the loop exits when the `__coroutine_state` pointer points to the noop-coroutine-state.

One strategy we can use here is to define a static instance of `__coroutine_state` that is designated as the noop-coroutine-state. The `std::noop_coroutine()` function can return a coroutine-handle that points to this object, and we can compare the `__coroutine_state` pointer to the address of that object to see if a particular coroutine handle is the noop-coroutine.

First let's define this special noop-coroutine-state object:

```

1  struct __coroutine_state {
2      using __resume_fn = __coroutine_state* (__coroutine_state*);
3      using __destroy_fn = void (__coroutine_state*);
4
5      __resume_fn* __resume;
6      __destroy_fn* __destroy;
7
8      static __coroutine_state* __noop_resume(__coroutine_state* state) noexcept {
9          return state;
10     }
11
12     static void __noop_destroy(__coroutine_state*) noexcept {}
13
14     static const __coroutine_state __noop_coroutine;
15 };
16
17 inline const __coroutine_state __coroutine_state::__noop_coroutine{
18     &__coroutine_state::__noop_resume,
19     &__coroutine_state::__noop_destroy
20 };

```

Then we can implement the `std::coroutine_handle<noop_coroutine_promise>` specialisation.

```

1  namespace std
2  {
3      struct noop_coroutine_promise {};
4
5      using noop_coroutine_handle = coroutine_handle<noop_coroutine_promise>;
6
7      noop_coroutine_handle noop_coroutine() noexcept;
8
9      template<>
10     class coroutine_handle<noop_coroutine_promise> {
11     public:
12         constexpr coroutine_handle(const coroutine_handle&) noexcept = default;
13         constexpr coroutine_handle& operator=(const coroutine_handle&) noexcept = default;
14
15         constexpr explicit operator bool() noexcept { return true; }
16
17         constexpr friend bool operator==(coroutine_handle, coroutine_handle) noexcept {
18             return true;
19         }
20     };

```

```

21     operator coroutine_handle<void>() const noexcept {
22         return coroutine_handle<void>::from_address(address());
23     }
24
25     noop_coroutine.promise& promise() const noexcept {
26         static noop_coroutine.promise promise;
27         return promise;
28     }
29
30     constexpr void resume() const noexcept {}
31     constexpr void destroy() const noexcept {}
32     constexpr bool done() const noexcept { return false; }
33
34     constexpr void* address() const noexcept {
35         return const_cast<__coroutine_state*>(&__coroutine_state::__noop_coroutine);
36     }
37
38     private:
39
40         friend noop_coroutine_handle noop_coroutine() noexcept {
41             return {};
42         }
43     };
44 }
```

And we can update `coroutine_handle::resume()` to exit when the noop-coroutine-state is returned.

```

1 void std::coroutine_handle<void>::resume() const {
2     __coroutine_state* s = state_;
3     do {
4         s = s->__resume(s);
5     } while (s != &__coroutine_state::__noop_coroutine);
6 }
```

And finally, we can update our `__g_resume()` function to now return the `__coroutine_state*`.

This just involves updating the signature and replacing:

```
coro_to_resume = ...;
goto resume_coro;
```

with

```
auto h = ...;
return static_cast<__coroutine_state*>(h.address());
```

and then at the very end of the function (after the `delete state;` statement) adding

```
return static_cast<__coroutine_state*>(std::noop_coroutine().address());
```

38.17 One last thing

Those with a keen eye may have noticed that the coroutine-state type `__g_state` is actually larger than it needs to be.

The data-members for the 4 temporary values each reserve storage for their respective values. However, the lifetimes of some of the temporary values do not overlap and so in theory we can save space in the coroutine-state by reusing the storage of an object for the next object after its lifetime has ended.

To be able to take advantage of this we can instead define the data-members in an anonymous union where appropriate.

Looking at the lifetimes of the temporary variables we have:

- `__tmp1` - exists only within `co_await promise.initial_suspend();` statement
- `__tmp2` - exists only within `int fx = co_await f(x);` statement
- `__tmp3` - exists only within `int fx = co_await f(x);` statement - nested inside lifetime of `__tmp2`
- `__tmp4` - exists only within `co_await promise.final_suspend();` statement

Since lifetimes of `__tmp2` and `__tmp3` overlap we must place them in a struct together as they both need to exist at the same time.

However, the `__tmp1` and `__tmp4` members do not have lifetimes that overlap and so they can be placed together in an anonymous `union`.

Thus we can change our data-member definition to:

```

1  struct __g_state : __coroutine_state_with_promise<__g_promise_t> {
2      __g_state(int&& x);
3      ~__g_state();
4
5      int __suspend_point = 0;
6      int x;
7
8      struct __scope1 {
9          manual_lifetime<task> __tmp2;
10         manual_lifetime<task::awaiter> __tmp3;
11     };
12
13     union {
14         manual_lifetime<std::suspend_always> __tmp1;
15         __scope1 __s1;
16         manual_lifetime<task::promise_type::final_awaiter> __tmp4;
17     };
18 }
```

Then, because the `__tmp2` and `__tmp3` variables are now nested inside the `__s1` object, we need to update references to them to now be e.g. `state->__s1.tmp2`. But otherwise the rest of the code stays the same.

This should save an additional 16 bytes of the coroutine-state size as we no longer need extra storage + padding for the `__tmp1` and `__tmp4` data-members - which would otherwise be padded to the size of a pointer, despite being empty types.

38.18 Tying it all together

Ok, so the final code we have generated for the coroutine function:

```
task g(int x) {
    int fx = co_await f(x);
    co_return fx * fx;
}
```

is the following:

```
1 //////
2 // The coroutine promise-type
3
4 using __g_promise_t = std::coroutine_traits<task, int>::promise_type;
5
6 __coroutine_state* __g_resume(__coroutine_state* s);
7 void __g_destroy(__coroutine_state* s);
8
9 //////
10 // The coroutine-state definition
11
12 struct __g_state : __coroutine_state_with.promise<__g_promise_t> {
13     __g_state(int&& x)
14     : x(static_cast<int&&>(x)) {
15         // Initialise the function-pointers used by coroutine_handle methods.
16         this->__resume = &__g_resume;
17         this->__destroy = &__g_destroy;
18
19         // Use placement-new to initialise the promise object in the base-class
20         // after we've initialised the argument copies.
21         ::new ((void*)std::addressof(this->__promise))
22             __g_promise_t(construct.promise<__g_promise_t>(this->x));
23     }
24
25     ~__g_state() {
26         this->__promise.~__g_promise_t();
27     }
28
29     int __suspend_point = 0;
```

```

30
31     // Argument copies
32     int x;
33
34     // Local variables/temporaries
35     struct __scope1 {
36         manual_lifetime<task> __tmp2;
37         manual_lifetime<task::awaiter> __tmp3;
38     };
39
40     union {
41         manual_lifetime<std::suspend_always> __tmp1;
42         __scope1 __s1;
43         manual_lifetime<task::promise_type::final_awaiter> __tmp4;
44     };
45 };
46
47 //////
48 // The "ramp" function
49
50 task g(int x) {
51     std::unique_ptr<__g_state> state(new __g_state(static_cast<int&&>(x)));
52     decltype(auto) return_value = state->__promise.get_return_object();
53
54     state->__tmp1.construct_from([&]() -> decltype(auto) {
55         return state->__promise.initial_suspend();
56     });
57     if (!state->__tmp1.get().await_ready()) {
58         state->__tmp1.get().await_suspend(
59             std::coroutine_handle<__g_promise_t>::from_promise(state->__promise));
60         state.release();
61         // fall through to return statement below.
62     } else {
63         // Coroutine did not suspend. Start executing the body immediately.
64         __g_resume(state.release());
65     }
66     return return_value;
67 }
68
69 //////
70 // The "resume" function
71

```

```

72  __coroutine_state* __g_resume(__coroutine_state* s) {
73      auto* state = static_cast<__g_state*>(s);
74
75      try {
76          switch (state->__suspend_point) {
77              case 0: goto suspend_point_0;
78              case 1: goto suspend_point_1; // <-- add new jump-table entry
79              default: std::unreachable();
80          }
81
82      suspend_point_0:
83      {
84          destructor_guard tmp1_dtor{state->__tmp1};
85          state->__tmp1.get().await_resume();
86      }
87
88      // int fx = co_await f(x);
89      {
90          state->__s1.__tmp2.construct_from([&] {
91              return f(state->x);
92          });
93          destructor_guard tmp2_dtor{state->__s1.__tmp2};
94
95          state->__s1.__tmp3.construct_from([&] {
96              return static_cast<task&&>(state->__s1.__tmp2.get()).operator co_await();
97          });
98          destructor_guard tmp3_dtor{state->__s1.__tmp3};
99
100         if (!state->__s1.__tmp3.get().await_ready()) {
101             state->__suspend_point = 1;
102
103             auto h = state->__s1.__tmp3.get().await_suspend(
104                 std::coroutine_handle<__g_promise_t>::from_promise(state->__promise));
105
106             // A coroutine suspends without exiting scopes.
107             // So cancel the destructor-guards.
108             tmp3_dtor.cancel();
109             tmp2_dtor.cancel();
110
111             return static_cast<__coroutine_state*>(h.address());
112         }
113     }

```

```

114     // Don't exit the scope here.
115     // We can't 'goto' a label that enters the scope of a variable with a
116     // non-trivial destructor. So we have to exit the scope of the destructor
117     // guards here without calling the destructors and then recreate them after
118     // the `suspend_point_1` label.
119     tmp3_dtor.cancel();
120     tmp2_dtor.cancel();
121 }
122
123 suspend_point_1:
124     int fx = [&]() -> decltype(auto) {
125         destructor_guard tmp2_dtor{state->__s1.__tmp2};
126         destructor_guard tmp3_dtor{state->__s1.__tmp3};
127         return state->__s1.__tmp3.get().await_resume();
128     }();
129
130     // co_return fx * fx;
131     state->__promise.return_value(fx * fx);
132     goto final_suspend;
133 } catch (...) {
134     state->__promise.unhandled_exception();
135     goto final_suspend;
136 }
137
138 final_suspend:
139     // co_await promise.final_suspend
140 {
141     state->__tmp4.construct_from([&]() noexcept {
142         return state->__promise.final_suspend();
143     });
144     destructor_guard tmp4_dtor{state->__tmp4};
145
146     if (!state->__tmp4.get().await_ready()) {
147         state->__suspend_point = 2;
148         state->__resume = nullptr; // mark as final suspend-point
149
150         auto h = state->__tmp4.get().await_suspend(
151             std::coroutine_handle<__g_promise_t>::from_promise(state->__promise));
152
153         tmp4_dtor.cancel();
154         return static_cast<__coroutine_state*>(h.address());
155     }

```

```
156
157     state->__tmp4.get().await_resume();
158 }
159
160 // Destroy coroutine-state if execution flows off end of coroutine
161 delete state;
162
163 return static_cast<__coroutine_state*>(std::noop_coroutine().address());
164 }
165
166 //////
167 // The "destroy" function
168
169 void __g_destroy(__coroutine_state* s) {
170     auto* state = static_cast<__g_state*>(s);
171
172     switch (state->__suspend_point) {
173     case 0: goto suspend_point_0;
174     case 1: goto suspend_point_1;
175     case 2: goto suspend_point_2;
176     default: std::unreachable();
177 }
178
179 suspend_point_0:
180     state->__tmp1.destroy();
181     goto destroy_state;
182
183 suspend_point_1:
184     state->__s1.__tmp3.destroy();
185     state->__s1.__tmp2.destroy();
186     goto destroy_state;
187
188 suspend_point_2:
189     state->__tmp4.destroy();
190     goto destroy_state;
191
192 destroy_state:
193     delete state;
194 }
```

For a fully compilable version of the final code, see: <https://godbolt.org/z/xaj3Yxabn>

This concludes the 5-part series on understanding the mechanics of C++ coroutines.

This is probably more information than you ever wanted to know about coroutines, but hopefully it

helps you to understand what's going on under the hood and demystifies them just a bit.

Thanks for making it through to the end!

Until next time, Lewis.

Chapter 39

Back Porting C++20 Coroutines to C++14 – A Tutorial

• Peter Rindal 📅 2022-01-24 🔖 ★★★

This post gives another perspective on the C++20 coroutines by presenting a small library¹ that emulates them using C++14 and a couple macros to aid readability. For example, the following two functions `g20()`, `g14()` both implement the same coroutine functionality. While the C++20 version is clearly a bit simpler, both are functionally equivalent and sufficiently readable. Next, we will unpack what the ‘‘coroutine transformation’’ is and how it can [partially] be implemented without specialized compiler support.

```
1 my_future<int> h();  
2  
3 // C++20 version  
4 my_future<int> g20() {  
5     int i = co_await h();  
6     if (i == 42)  
7         i = 0;  
8     co_return i;  
9 }  
10  
11 // C++14 version  
12 my_future<int> g14() {  
13     // Specify the return type and lambda capture and local variables.  
14     CO_BEGIN(my_future<int>, i = int{});  
15  
16     // assign the result of h() to i  
17     CO_AWAIT(i, h());  
18     if (i == 42)  
19         i = 0;  
20     CO_RETURN(i);
```

¹<http://github.com/ladnir/macoro>

```

21
22     CO_END();
23 }
```

Due to certain limitations of macros, etc., all local variables that span a CO_XXXX macro must be “lambda captured” in the CO_BEGIN macro. We will discuss this more below but CO_BEGIN must be passed the return type followed by the body of a lambda capture as if the rest of the code is in a lambda that can not have local variables. For example, CO_END(my_future<int>, i = int{}) loosely gets translated into

```
[i = int{}] ()->my_future<int> {
    //<body-of-the-coroutine>
}
```

This likely gives the reader a hint at how we will port C++20 coroutines back to C++14.

39.1 Coroutine Usage

First, let us review some basic coroutine functionality. If the reader is familiar with coroutines feel free to skip to the next section. Coroutines can be thought of as a stateful function that can “pause” its execution and then be resumed later by the caller or someone else.

39.1.1 co_await

There are a couple of situations where a coroutine might be paused, or “suspend” in coroutine terminology. The most common is when the coroutine executes a `co_await` call of some awaitable (e.g. some other coroutine) and the result is not ready yet. For example, in the functions above we call `co_await h()`. The function `h()` returns a `my_future<int>` which implements the interface of an “awaitable” with value type `int`. This interface allows the compiler to auto generate some code that determines if the underlying value is ready now or if the current coroutine should suspend while we wait for `h()` to complete.

To make the example a bit more concrete, `h()` could be fetching a value from some server on the internet which is slow. While we wait, we want to pause this coroutine and allow the caller to go do something else. Once `h()` finishes, our coroutine can be resumed either by `h()`, by some other thread or we can be notified in some way and manually resume the coroutine. The how/when/who resumes the coroutine can be customized by our return type `my_future<int>` and what we are `co_awaiting` on.

If the underlying `int` that is provided by `h()` is ready immediately, our coroutine does not need to suspend. Whenever the underlying value is ready, either immediately after the call to `h()` or when the coroutine is resumed, execution of the coroutine continues. The next thing to be executed is compiler generated code that passes the value held in the `my_future<int>` to the current coroutine, i.e. assigns it to `int i` in our examples. More details on this compiler generated code later. Finally, execution of the coroutine continues with the next statement after the `co_await`.

39.1.2 co_yield

Another example of when a coroutine can be suspended is with the `co_yield` keyword. The exact semantics of this is customizable but typically it is used to suspend the current coroutine and pass a value

back to the caller. This is similar to returning a value but we are allowed to resume the coroutine later. Typically, the caller will use this value and then resume the coroutine which can `co_yield` another value back to the caller or run to completion. A classic example of this is to lazily generate a range of values. For example, `range20`, `range14` both return a `my_range<int>` which allows the caller to iterate over the values that are `co_yield` out of the coroutine. This example prints the string `2,3,4,5,6,7,8,9,.`

```

1 my_range<int> range20(int begin, int end) {
2     for (int i = begin; i < end; ++i)
3         co_yield i;
4 }
5 my_range<int> range14(int begin, int end) {
6     CO_BEGIN(my_range<int>, begin, end, i = int{});
7     for (i = begin; i < end; ++i)
8         CO_YIELD(i);
9     CO_END();
10 }
11
12 void example() {
13     for (int i : range14(2, 10))
14         std::cout << i << ',';
15 }
```

Importantly, the return type of the coroutine, in this case `my_future<int>`, must implement a certain interface in order for the coroutine body to interact with it. In brief, this interface consists of the return type having an associated `promise_type` which implements certain functions. For `co_yield <expr>`; statements, the `<expr>`; value is forwarded to `promise_type::yield_value(<expr>)` which, in our example, somehow forwards it to the return value `my_range<int>` so that the caller can consume it. Similar “customization points” are required for determining how the other situations are handled.

39.1.3 co_return

To return a value from a coroutine we must use `co_return` instead of normal `return`. This special return keyword helps the compiler know that the current function is a coroutine as opposed to a normal function. In particular, from the function signature `my_future<int> g20()` there is no way to tell if the body of the function is coroutine or a normal function that just happens to return a `my_future<int>`.

More generally, whenever the compiler sees at least one `co_await`, `co_yield` or `co_return` keyword, it will convert the function body into a coroutine by performing the “coroutine transformation.” Importantly, this transformation is just an implementation detail. The caller does not know or care if `my_future<int> g20()` is a coroutine or implements the required logic in some other way.

Next, we will discuss precisely that, how to implement the coroutine transformation manually and then introduce the macros you see above to aid readability.

39.2 The Coroutine Transformation

At the core of the coroutine specification lies a relatively mechanical transformation. Every coroutine function is effectively transformed into two functions. The first (outer function) performs the following:

- Constructs the so-called promise type `<function-return-type>::promise_type promise`. The role of the promise type is to decide when to start and/or resume the coroutine and what to do when it yields a value or returns.
- Construct the return type `<function-return-type>` which in our first example is `my_future<int>`. This is obtained by calling `promise.get_return_object()`. This will be what is returned to the caller.
- Query the `promise_type` and decide if the coroutine body should be started now (before returning to the caller) or later (possibly by the caller). This is determined by the result of `promise.init_ial_suspend()`. If we should start now then we call the second function which contains the user code.
- Finally, we return the result of `promise.get_return_object()` to the caller.

The second function (inner function) of the transformation will contain the actual body of the coroutine that the user wrote. Each time `co_await`, `co_yield` or `co_return` is seen, code is inserted to perform the required logic and allow the coroutine to be suspended/resumed at this location. These are called suspension points.

Below is some C++ inspired pseudocode that loosely implements the required logic which we explain next. The outer function `g_inlined` is intended to have the same behavior as `g20()`, `g14()` above.

```

1 my_future<int> g_inlined()
2 {
3     // Construct the coroutine frame and promise
4     struct coroutine_frame {
5         my_future<int>::promise_type promise;
6         std::function<void(void)> inner_fn;
7         void* suspend_address = &InitialSuspend;
8     };
9     auto frame = new coroutine_frame;
10
11    // define the inner function
12    frame->inner_fn = [frame]() {
13        // jump to wherever we left off.
14        goto frame->suspend_address;
15    InitialSuspend:
16        // <---- beginning of user code. ---->
17
18        // int i = co_await h();
19        auto awaiter_h = h();
```

```

20     if (!awaiter_h.await_ready()) {
21         frame->suspend_address = &Suspend_1;
22         awaiter_h.await_suspend(frame);
23         return;
24     }
25 Suspend_1:
26     int i = awaiter_h.await_resume();
27
28     // normal code
29     if (i == 42)
30         i = 0;
31
32     // co_return i;
33     frame->suspend_address = &FinalSuspend;
34     frame->promise.return_value(i);
35
36     // <---- end of user code. ---->
37     // final suspend
38     if (!promise.final_suspend().await_ready()) {
39         frame->suspend_address = &FinalSuspend;
40         return;
41     }
42 FinalSuspend:
43     delete frame;
44 }
45
46 my_future<int> ret = frame->promise.get_return_object();
47
48 // call the inner function if we are ready.
49 if (frame->promise.initial_suspend().await_ready())
50     frame->inner_fn();
51
52 return ret;
53 }
```

First, the so-called `coroutine_frame` is allocated. The frame will contain the `promise_type` and space for the local variables used in the inner function, which for clarity is being managed using `std::function<void(void)>`.

This inner function can be called many times with the desired logic being that the function should resume from wherever it left off. The code above achieves this using a hypothetical syntax for storing the addresses of `goto` labels. In particular, each time we enter the inner function we will perform `goto` `suspend_address` where `suspend_address` is the location in the function that is just after where we suspended last.

The first time we enter the function `suspend_address` will have the initial value of `InitialSuspend`: and begin executing the user code. At some point the execution will hit a `co_*` statement, in this case `int i = co_await h()`. This statement will be transformed as shown above to first query if the value return by `auto awaiter_h = h()` is ready to be consumed. This is achieved by calling `awaiter_h.await_ready()`.

If `awaiter_h.await_ready()` returns `true` then we need to suspend the coroutine. This is achieved by updating the `suspend_address` to be just after the current `co_await` statement and calling `awaiter_h.await_suspend(frame)` which gives `awaiter_h` the responsibility to resume this coroutine whenever the value does become available. Finally, we return from the inner function, in this case back to the outer function and then to initial caller. Through some other mechanism, `awaiter_h` will eventually have a value (or exception) and the inner function will be resumed. This time the `goto` `suspend_address` statement will jump us back to just after the `return` that was taken last time, i.e. `Suspend_1`.

Alternatively, `awaiter_h.await_ready()` could have returned `true` and the program would similarly be at the `Suspend_1` location. Regardless, the result of `co_await h()` statement is ready and can be obtained by calling `awaiter_h.await_resume()`.

The `co_return i` statement is handled by calling `promise.return_value(i);` and then immediately proceeding to the so-called `final_suspend`. The role of `final_suspend` is to determine if the current `coroutine_frame` which contains the user's `promise_type` is safe to be deleted. Often it is preferred to not delete the frame at this time and instead allow the caller/consumer to delete the frame once they have obtained the result, i.e. `i` in this case.

39.3 Issues and Solutions

The code given above should give you an intuition about what the compiler is doing. However, there are several issues in emulating as described above.

39.3.1 goto suspect_point

First, is that standard C++ does not allow you to take the address of a `goto` label. To address this the `goto` statements can be replaced with a `switch` statement. The function body would then look like

```

1  {
2      switch (frame->suspend_point)
3      {
4          case SuspensionPoint::InitialSuspend:
5              // int i = co_await h();
6              auto awaiter_h = h();
7              if (!awaiter_h.await_ready()) {
8                  frame->suspend_point = (SuspensionPoint)1;
9                  awaiter_h.await_suspend(frame);
10                 return;
11             }
12             case (SuspensionPoint)1:
13                 int i = awaiter_h.await_resume();

```

```

14
15     //...
16
17     case SuspensionPoint::FinalSuspend:
18         delete frame;
19     }
20 }
```

Using a switch statement is almost functionally equivalent. The main downside is that users can not use switch statements across `co_*` keywords (looking forward `CO_*` macros). Interestingly, switch statements allow for `case` locations that are in a nested scope, e.g. using `can_suspend` in the middle of a for-loop and the switch statement can jump back to it upon resume.

39.3.2 Local variables

The second issue regards saving the state of local variables between suspend and resume calls. For example, if before `co_await h()` the user has constructed some local variable and we suspended for `h()`, then when the function is resumed that local variable would be uninitialized.

The compiler handles this in the transformation by moving any declaration of local variables into the `coroutine_frame` as opposed to being the body of our inner function. They are then destroyed when the coroutine completes.

While one can manually perform the transformation, we will instead consider an alternative solution. In particular, we will place/require all local variables to be declared in the lambda capture. While this results in slightly different semantics, it will enable us to write the `CO_BEGIN` macro.

A similar issue regards the lifetime of the awainer when the coroutine is suspended, i.e. `awaiter_h` in the example above. This can be resolved by storing the awainer either in a fixed sized buffer in `coroutine_frame` or allocating it on the heap if it is too large to fit in the buffer.

39.3.3 std::coroutine_handle<promise_type>

The final issue regards compatibility with C++20 coroutines. Ideally we can choose to implement our coroutines manually using the pattern outlined here or use the C++20 keywords. However, the primary barrier in achieving this is that C++20 defines `<awaiter>.await_suspend` as taking a `std::coroutine_handle<promise_type>` as input while our example takes `<function-name>::coroutine_frame*`.

Intuitively, these two parameters represent the same thing. Both are a pointer to the coroutine frame and provide a way for the awainer to resume this coroutine once the thing being awaited becomes ready. However, the way we resume the `<function-name>::coroutine_frame*` vs a C++20 coroutine are concretely different. Moreover, there is no way to convert our `<function-name>::coroutine_frame*` into a `std::coroutine_handle<promise_type>` since we can not modify that type.

This issue could be resolved by giving `promise_type` another customization point which `std::coroutine_handle<promise_type>` interacts with and then dispatches to the correct method for resuming to coroutine. However, this is not part of the standard.

Instead, we will define our own `macoro::coroutine_handle<promise_type>` which can hold either a `std::coroutine_handle<promise_type>` or a `<function-name>::coroutine_frame*`. As such,

our coroutines will only work with a `promise_type` that can take a `macoro::coroutine_handle<*>` as input to `<awaiter>.await_suspend(...)`.

This is somewhat unfortunate because thirdparty awaitable types which are intended for C++20 will most likely not work out of the box. However, this is probably unavoidable given that they will likely take other dependencies to C++20.

39.3.4 Allocations

The current strategy requires several allocations. First, we must allocate the `coroutine_frame`. C++20 defines a way to allow the user to control where this allocation happens. A description of it can be found in Lewis Baker’s excellent blog posts². A similar strategy can be applied to our framework with some modifications.

Another allocation likely occurs for the `std::function<void(void)>`. However, this is not required. We can instead define the function

```
template<typename lambda_type, typename promise_type>
coroutine_frame<lambda_type, promise_type>* makeFrame(lambda_type&& inner_fn);
```

that returns a frame that stores the inner function with type `lambda_type` inside the `coroutine_frame`. Then the inner function will take a pointer to the frame as its only parameter.

The final allocation is with respect to the storage of any awaiters that are produced during an active `co_await`. As suggested previously, these can be partially handled by having an additional buffer within the `coroutine_frame`. However, it is still possible that the coroutine body awaits an awaiter that does not fit in the buffer and then an allocation would occur. However, it is possible for this to result in a compile time error. This can then be combined with allowing the buffer size to be a template parameter, i.e. an input to `CO_BEGIN`.

Combining these we could obtain a coroutine that is guaranteed to not to allocation any memory, something that C++20 coroutines can not currently achieve 100 percent of the time.

39.4 Macros for Readability

Using the pattern above we can define the macros shown in the beginning. A prototype of these ideas can be found in the macororepo. Below is a simplified definition of the main macros. In summary, `CO_BEGIN` begins the call to `makeFrame` that allocates the frame with the inner function being stored inline. The closing brace of the lambda and `makeFrame` function is in `CO_END`. The first time we enter the inner function we finish the initial suspend (which was started in the outer function in `CO_END`) by calling `await_resume` and destroying the awaiter.

`CO_AWAIT` first computes the type of the awaiter and then constructs it within the frame (or on the heap). It then performs the `await_ready/await_suspend/await_resume` procedure. If it does suspend, it updates the suspend point with the line number.

`CO_END` performs the catch block, passing any exception to the `promise` and then performs the final suspend. This completes the definition of the inner function. `CO_END` then completes the outer function by constructing the return object and starting the initial suspend. If ready we start the coroutine by calling the inner function and then return the return type.

²<https://lewissbaker.github.io/>

39.4.1 CO_BEGIN(return_type, ...)

```

1  using promise_type = return_type::promise_type;
2  using Handle = coroutine_handle<promise_type>;
3  // make frame with inline body.
4  auto frame = makeFrame<promise_type>(
5      [__VA_ARGS__](FrameBase<promise_type>* frame) mutable {
6
7      promise_type& promise = frame->promise;
8      try {
9          // jump to where we suspended
10         switch (frame->getSuspendPoint())
11         {
12             case SuspensionPoint::InitialSuspend:
13             { // ----- initial suspend continued -----
14                 using Awaiter = typename awaiter_for<promise_type,
15                     decltype(promise.initial_suspend())>;
16                 frame->getAwaiter<Awaiter>().await_resume();
17                 frame->destroyAwaiter<Awaiter>();
18             } // ----- end of initial suspend -----
19             // ----- beginning of user code -----

```

39.4.2 CO_AWAIT(RETURN_VAL, EXPR)

```

1  {
2      using Awaiter = awaiter_for<promise_type, decltype(EXPR)>;
3      {
4          auto& awaiter = frame->constructAwaiter(EXPR);
5          if (!awaiter.await_ready()) {
6              // suspend-coroutine
7              frame->setSuspendPoint((SuspensionPoint)__LINE__);
8
9              // call awaiter.await_suspend(). If it's void return, then return true.
10             if (await_suspend(awaiter, Handle::from_promise(promise)))
11                 return;
12         }
13     }
14     // resume-point, __LINE__ will be the same in macro.
15     case SuspensionPoint(__LINE__):
16         RETURN_VAL = frame->getAwaiter<Awaiter>().await_resume();
17         frame->destroyAwaiter<Awaiter>();
18     }

```

39.4.3 CO_END()

```

1      // ----- end of user code -----
2      default:
3      }
4      }
5      catch (...) {
6          frame->promise.unhandled_exception();
7      }
8
9      // ----- beginning of final suspend -----
10     using Awaiter = typename awaiter_for<promise_type, decltype(promise.final_suspend())>;
11     auto& awaiter = frame->constructAwaiter(promise.final_suspend());
12     auto handle = Handle::from_promise(promise);
13     if (!awaiter.await_ready()) {
14         // suspend-coroutine
15         frame->setSuspendPoint(SuspensionPoint::FinalSuspend);
16         // call awaiter.await_suspend(). If it's void return, then return true.
17         if (await_suspend(awaiter, handle))
18             return;
19     }
20     frame->getAwaiter<Awaiter>().await_resume();
21     frame->destroyAwaiter<Awaiter>();
22     handle.destroy();
23     // ----- end of final suspend -----
24 };
25
26     promise_type& promise = frame->promise;
27     auto ret = promise.get_return_object();
28
29     // ----- beginning of initial suspend -----
30     using Awaiter = awaiter_for<promise_type, decltype(promise.initial_suspend())>;
31     auto& awaiter = frame->constructAwaiter(promise.initial_suspend());
32     auto handle = Handle::from_promise(promise);
33     if (!awaiter.await_ready())
34     {
35         // suspend-coroutine
36         frame->setSuspendPoint(SuspensionPoint::InitialSuspend);
37         //call awaiter.await_suspend(). If it's void return, then return true
38         if (await_suspend(awaiter, handle))
39         return ret;
40     }
41

```

```
42  /*begin coroutine*/
43  (*frame)(frame);
44  return ret;
```

Chapter 40

Debugging C++ Coroutines

👤 The Clang Team 📅 2022-07 🏷 ★★★★

40.1 Introduction

For performance and other architectural reasons, the C++ Coroutines feature in the Clang compiler is implemented in two parts of the compiler. Semantic analysis is performed in Clang, and Coroutine construction and optimization takes place in the LLVM middle-end.

However, this design forces us to generate insufficient debugging information. Typically, the compiler generates debug information in the Clang frontend, as debug information is highly language specific. However, this is not possible for Coroutine frames because the frames are constructed in the LLVM middle-end.

To mitigate this problem, the LLVM middle end attempts to generate some debug information, which is unfortunately incomplete, since much of the language specific information is missing in the middle end.

This document describes how to use this debug information to better debug coroutines.

40.2 Terminology

Due to the recent nature of C++20 Coroutines, the terminology used to describe the concepts of Coroutines is not settled. This section defines a common, understandable terminology to be used consistently throughout this document.

40.2.1 coroutine type

A coroutine function is any function that contains any of the Coroutine Keywords `co_await`, `co_yield`, or `co_return`. A coroutine type is a possible return type of one of these coroutine functions. Task and Generator are commonly referred to coroutine types.

40.2.2 coroutine

By technical definition, a coroutine is a suspendable function. However, programmers typically use coroutine to refer to an individual instance. For example:

```
std::vector<Task> Coros; // Task is a coroutine type.
for (int i = 0; i < 3; i++)
    Coros.push_back(CoroTask()); // CoroTask is a coroutine function, which
                                // would return a coroutine type 'Task'.
```

In practice, we typically say “*Coros* contains 3 coroutines” in the above example, though this is not strictly correct. More technically, this should say “*Coros* contains 3 coroutine instances” or “*Coros* contains 3 coroutine objects.”

In this document, we follow the common practice of using *coroutine* to refer to an individual *coroutine instance*, since the terms *coroutine instance* and *coroutine object* aren’t sufficiently defined in this case.

40.2.3 coroutine frame

The C++ Standard uses *coroutine state* to describe the allocated storage. In the compiler, we use *coroutine frame* to describe the generated data structure that contains the necessary information.

40.3 The structure of coroutine frames

The structure of coroutine frames is defined as:

```
struct {
    void (*__r)(); // function pointer to the `resume` function
    void (*__d)(); // function pointer to the `destroy` function
    promise_type; // the corresponding `promise_type`
    ... // Any other needed information
}
```

In the debugger, the function’s name is obtainable from the address of the function. And the name of *resume* function is equal to the name of the coroutine function. So the name of the coroutine is obtainable once the address of the coroutine is known.

40.4 Print promise_type

Every coroutine has a *promise_type*, which defines the behavior for the corresponding coroutine. In other words, if two coroutines have the same *promise_type*, they should behave in the same way. To print a *promise_type* in a debugger when stopped at a breakpoint inside a coroutine, printing the *promise_type* can be done by:

```
print __promise
```

It is also possible to print the *promise_type* of a coroutine from the address of the coroutine frame. For example, if the address of a coroutine frame is `0x416eb0`, and the type of the *promise_type* is `task_::promise_type`, printing the *promise_type* can be done by:

```
print (task_::promise_type)*(0x416eb0+0x10)
```

This is possible because the `promise_type` is guaranteed by the ABI to be at a 16 bit offset from the coroutine frame.

Note that there is also an ABI independent method:

```
print std::coroutine_handle<task::promise_type>::from_address((void*)0x416eb0).promise()
```

The functions `from_address(void*)` and `promise()` are often small enough to be removed during optimization, so this method may not be possible.

40.5 Print coroutine frames

LLVM generates the debug information for the coroutine frame in the LLVM middle end, which permits printing of the coroutine frame in the debugger. Much like the `promise_type`, when stopped at a breakpoint inside a coroutine we can print the coroutine frame by:

```
print __coro_frame
```

Just as printing the `promise_type` is possible from the coroutine address, printing the details of the coroutine frame from an address is also possible:

```
1 (gdb) # Get the address of coroutine frame
2 (gdb) print/x *0x418eb0
3 $1 = 0x4019e0
4 (gdb) # Get the linkage name for the coroutine
5 (gdb) x 0x4019e0
6 0x4019e0 <_ZL9coro_taski>: 0xe5894855
7 (gdb) # Turn off the demangler temporarily to avoid the debugger misunderstanding the name.
8 (gdb) set demangle-style none
9 (gdb) # The coroutine frame type is 'linkage_name.coro_frame_ty'
10 (gdb) print (_ZL9coro_taski.coro_frame_ty)*(0x418eb0)
11 $2 = {__resume_fn = 0x4019e0 <coro_task(int)>, __destroy_fn = 0x402000 <coro_task(int)>,
12 __promise = {...}, ...}
```

The above is possible because:

1. The name of the debug type of the coroutine frame is the `linkage_name`, plus the `.coro_frame_ty` suffix because each coroutine function shares the same coroutine type.
2. The coroutine function name is accessible from the address of the coroutine frame.

The above commands can be simplified by placing them in debug scripts.

40.5.1 Examples to print coroutine frames

The print examples below use the following definition:

```
1 #include <coroutine>
2 #include <iostream>
3
```

```
4  struct task{
5      struct promise_type {
6          task get_return_object() {
7              return std::coroutine_handle<promise_type>::from_promise(*this);
8          }
9          std::suspend_always initial_suspend() { return {}; }
10         std::suspend_always final_suspend() noexcept { return {}; }
11         void return_void() noexcept {}
12         void unhandled_exception() noexcept {}
13
14     int count = 0;
15     };
16
17     void resume() noexcept {
18         handle.resume();
19     }
20
21     task(std::coroutine_handle<promise_type> hdl) : handle(hdl) {}
22
23     ~task() {
24         if (handle)
25             handle.destroy();
26     }
27
28     std::coroutine_handle<> handle;
29 };
30
31 class await_counter : public std::suspend_always {
32     public:
33     template<class PromiseType>
34     void await_suspend(std::coroutine_handle<PromiseType> handle) noexcept {
35         handle.promise().count++;
36     }
37 };
38
39 static task coro_task(int v) {
40     int a = v;
41     co_await await_counter{};
42     a++;
43     std::cout << a << "\n";
44     a++;
45     std::cout << a << "\n";
46     a++;
47     std::cout << a << "\n";
48 }
```

```

46     co_await await_counter{};
47     a++;
48     std::cout << a << "\n";
49     a++;
50     std::cout << a << "\n";
51 }
52
53 int main() {
54     task t = coro_task(43);
55     t.resume();
56     t.resume();
57     t.resume();
58     return 0;
59 }
```

In debug mode (00 + g), the printing result would be:

```

1 {__resume_fn = 0x4019e0 <coro_task(int)>, __destroy_fn = 0x402000 <coro_task(int)>,
2   __promise = {count = 1}, v = 43, __coro_index = 1 '001',
3   struct_std__suspend_always_0 = {__int_8 = 0 '000'},
4   class_await_counter_1 = {__int_8 = 0 '000}, class_await_counter_2 =
5   {__int_8 = 0 '000}, struct_std__suspend_always_3 = {__int_8 = 0 '000}}
```

In the above, the values of v and a are clearly expressed, as are the temporary values for await_counter (class_await_counter_1 and class_await_counter_2) and std::suspend_always (struct_std__suspend_always_0 and struct_std__suspend_always_3). The index of the current suspension point of the coroutine is emitted as __coro_index. In the above example, the __coro_index value of 1 means the coroutine stopped at the second suspend point (Note that __coro_index is zero indexed) which is the first co_await await_counter{}; in coro_task. Note that the first initial suspend point is the compiler generated co_await promise_type::initial_suspend().

However, when optimizations are enabled, the printed result changes drastically:

```
{__resume_fn = 0x401280 <coro_task(int)>, __destroy_fn = 0x401390 <coro_task(int)>,
 __promise = {count = 1}, __int_32_0 = 43, __coro_index = 1 '001'}
```

Unused values are optimized out, as well as the name of the local variable a. The only information remained is the value of a 32 bit integer. In this simple case, it seems to be pretty clear that __int_32_0 represents a. However, it is not true.

An important note with optimization is that the value of a variable may not properly express the intended value in the source code. For example:

```

1 static task coro_task(int v) {
2     int a = v;
3     co_await await_counter{};
4     a++; // __int_32_0 is 43 here
5     std::cout << a << "\n";
```

```

6     a++; // __int_32_0 is still 43 here
7     std::cout << a << "\n";
8     a++; // __int_32_0 is still 43 here!
9     std::cout << a << "\n";
10    co_await await_counter{};
11    a++; // __int_32_0 is still 43 here!!
12    std::cout << a << "\n";
13    a++; // Why is __int_32_0 still 43 here?
14    std::cout << a << "\n";
15 }

```

When debugging step-by-step, the value of `__int_32_0` seemingly does not change, despite being frequently incremented, and instead is always 43. While this might be surprising, this is a result of the optimizer recognizing that it can eliminate most of the load/store operations. The above code gets optimized to the equivalent of:

```

1 static task coro_task(int v) {
2     store v to __int_32_0 in the frame
3     co_await await_counter{};
4     a = load __int_32_0
5     std::cout << a+1 << "\n";
6     std::cout << a+2 << "\n";
7     std::cout << a+3 << "\n";
8     co_await await_counter{};
9     a = load __int_32_0
10    std::cout << a+4 << "\n";
11    std::cout << a+5 << "\n";
12 }

```

It should now be obvious why the value of `__int_32_0` remains unchanged throughout the function. It is important to recognize that `__int_32_0` does not directly correspond to `a`, but is instead a variable generated to assist the compiler in code generation. The variables in an optimized coroutine frame should not be thought of as directly representing the variables in the C++ source.

40.6 Get the suspended points

An important requirement for debugging coroutines is to understand suspended points, which are where the coroutine is currently suspended and awaiting.

For simple cases like the above, inspecting the value of the `__coro_index` variable in the coroutine frame works well.

However, it is not quite so simple in really complex situations. In these cases, it is necessary to use the coroutine libraries to insert the line-number.

For example:

```

1 // For all the promise_type we want:
2 class promise_type {
3     ...
4     + unsigned line_number = 0xffffffff;
5 };
6
7 #include <source_location>
8
9 // For all the awaiter types we need:
10 class awaiter {
11     ...
12     template <typename Promise>
13     void await_suspend(std::coroutine_handle<Promise> handle,
14                         std::source_location sl = std::source_location::current()) {
15         ...
16         handle.promise().line_number = sl.line();
17     }
18 };

```

In this case, we use `std::source_location` to store the line number of the await inside the `promise_type`. Since we can locate the coroutine function from the address of the coroutine, we can identify suspended points this way as well.

The downside here is that this comes at the price of additional runtime cost. This is consistent with the C++ philosophy of “Pay for what you use” .

40.7 Get the asynchronous stack

Another important requirement to debug a coroutine is to print the asynchronous stack to identify the asynchronous caller of the coroutine. As many implementations of coroutine types store `std::coroutine_handle<>` continuation in the promise type, identifying the caller should be trivial. The continuation is typically the awaiting coroutine for the current coroutine. That is, the asynchronous parent.

Since the `promise_type` is obtainable from the address of a coroutine and contains the corresponding continuation (which itself is a coroutine with a `promise_type`), it should be trivial to print the entire asynchronous stack.

This logic should be quite easily captured in a debugger script.

40.7.1 Examples to print asynchronous stack

Here is an example to print the asynchronous stack for the normal task implementation.

```

1 // debugging-example.cpp
2 #include <coroutine>
3 #include <iostream>
4 #include <utility>

```

```
5
6 struct task {
7     struct promise_type {
8         task get_return_object();
9         std::suspend_always initial_suspend() { return {}; }
10
11    void unhandled_exception() noexcept {}
12
13    struct FinalSuspend {
14        std::coroutine_handle<> continuation;
15        auto await_ready() noexcept { return false; }
16        auto await_suspend(std::coroutine_handle<> handle) noexcept {
17            return continuation;
18        }
19        void await_resume() noexcept {}
20    };
21    FinalSuspend final_suspend() noexcept { return {continuation}; }
22
23    void return_value(int res) { result = res; }
24
25    std::coroutine_handle<> continuation = std::noop_coroutine();
26    int result = 0;
27};
28
29    task(std::coroutine_handle<promise_type> handle) : handle(handle) {}
30    ~task() {
31        if (handle)
32            handle.destroy();
33    }
34
35    auto operator co_await() {
36        struct Awaiter {
37            std::coroutine_handle<promise_type> handle;
38            auto await_ready() { return false; }
39            auto await_suspend(std::coroutine_handle<> continuation) {
40                handle.promise().continuation = continuation;
41                return handle;
42            }
43            int await_resume() {
44                int ret = handle.promise().result;
45                handle.destroy();
46                return ret;
47            }
48        };
49        return Awaiter();
50    }
51}
```

```

47         }
48     };
49     return Awaiter{std::exchange(handle, nullptr)};
50   }
51
52   int syncStart() {
53     handle.resume();
54     return handle.promise().result;
55   }
56
57 private:
58   std::coroutine_handle<promise_type> handle;
59 };
60
61 task promise_type::get_return_object() {
62   return std::coroutine_handle<promise_type>::from_promise(*this);
63 }
64
65 namespace detail {
66 template <int N>
67 task chain_fn() {
68   co_return N + co_await chain_fn<N - 1>();
69 }
70
71 template <>
72 task chain_fn<0>() {
73   // This is the default breakpoint.
74   __builtin_debugtrap();
75   co_return 0;
76 }
77 } // namespace detail
78
79 task chain() {
80   co_return co_await detail::chain_fn<30>();
81 }
82
83 int main() {
84   std::cout << chain().syncStart() << "\n";
85   return 0;
86 }

```

In the example, the `task` coroutine holds a `continuation` field, which would be resumed once the `task` completes. In another word, the `continuation` is the asynchronous caller for the `task`. Just like the

normal function returns to its caller when the function completes.

So we can use the continuation field to construct the asynchronous stack:

```
# debugging-helper.py
import gdb
from gdb.FrameDecorator import FrameDecorator

class SymValueWrapper():
    def __init__(self, symbol, value):
        self.sym = symbol
        self.val = value

    def __str__(self):
        return str(self.sym) + " = " + str(self.val)

    def get_long_pointer_size():
        return gdb.lookup_type('long').pointer().sizeof

    def cast_addr2long_pointer(addr):
        return gdb.Value(addr).cast(gdb.lookup_type('long').pointer())

    def dereference(addr):
        return long(cast_addr2long_pointer(addr).dereference())

class CoroutineFrame(object):
    def __init__(self, task_addr):
        self.frame_addr = task_addr
        self.resume_addr = task_addr
        self.destroy_addr = task_addr + get_long_pointer_size()
        self.promise_addr = task_addr + get_long_pointer_size() * 2
        # In the example, the continuation is the first field member of the promise_type.
        # So they have the same addresses.
        # If we want to generalize the scripts to other coroutine types, we need to be sure
        # the continuation field is the first member of promise_type.
        self.continuation_addr = self.promise_addr

    def next_task_addr(self):
        return dereference(self.continuation_addr)

class CoroutineFrameDecorator(FrameDecorator):
    def __init__(self, coro_frame):
        super(CoroutineFrameDecorator, self).__init__(None)
        self.coro_frame = coro_frame
```

```

        self.resume_func = dereference(self.coro_frame.resume_addr)
        self.resume_func_block = gdb.block_for_pc(self.resume_func)
        if self.resume_func_block == None:
            raise Exception('Not stackless coroutine.')
        self.line_info = gdb.find_pc_line(self.resume_func)

    def address(self):
        return self.resume_func

    def filename(self):
        return self.line_info.symtab.filename

    def frame_args(self):
        return [
            SymValueWrapper("frame_addr",
                cast_addr2long_pointer(self.coro_frame.frame_addr)),
            SymValueWrapper("promise_addr",
                cast_addr2long_pointer(self.coro_frame.promise_addr)),
            SymValueWrapper("continuation_addr",
                cast_addr2long_pointer(self.coro_frame.continuation_addr))
        ]

    def function(self):
        return self.resume_func_block.function.print_name

    def line(self):
        return self.line_info.line

    class StripDecorator(FrameDecorator):
        def __init__(self, frame):
            super(StripDecorator, self).__init__(frame)
            self.frame = frame
            f = frame.function()
            self.function_name = f

        def __str__(self, shift = 2):
            addr = "" if self.address() == None else '%#x' % self.address() + " in "
            location = "" if self.filename() == None else " at " +
                self.filename() + ":" + str(self.line())
            return addr + self.function() + " " +
                str([str(args) for args in self.frame_args()]) + location

```

```

class CoroutineFilter:
    def create_coroutine_frames(self, task_addr):
        frames = []
        while task_addr != 0:
            coro_frame = CoroutineFrame(task_addr)
            frames.append(CoroutineFrameDecorator(coro_frame))
            task_addr = coro_frame.next_task_addr()
        return frames

class AsyncStack(gdb.Command):
    def __init__(self):
        super(AsyncStack, self).__init__("async-bt", gdb.COMMAND_USER)

    def invoke(self, arg, from_tty):
        coroutine_filter = CoroutineFilter()
        argv = gdb.string_to_argv(arg)
        if len(argv) == 0:
            try:
                task = gdb.parse_and_eval('__coro_frame')
                task = int(str(task.address), 16)
            except Exception:
                print ("Can't find __coro_frame in current context.\n" +
                      "Please use `async-bt` in stackless coroutine context.")
            return
        elif len(argv) != 1:
            print("usage: async-bt <pointer to task>")
            return
        else:
            task = int(argv[0], 16)

        frames = coroutine_filter.create_coroutine_frames(task)
        i = 0
        for f in frames:
            print '#'+ str(i), str(StripDecorator(f))
            i += 1
        return

    AsyncStack()

class ShowCoroFrame(gdb.Command):
    def __init__(self):
        super(ShowCoroFrame, self).__init__("show-coroframe", gdb.COMMAND_USER)

```

```

def invoke(self, arg, from_tty):
    argv = gdb.string_to_argv(arg)
    if len(argv) != 1:
        print("usage: show-coroutine-frame <address of coroutine frame>")
        return

    addr = int(argv[0], 16)
    block = gdb.block_for_pc(long(cast_addr2long_pointer(addr)).dereference())
    if block == None:
        print "block " + str(addr) + " is none."
        return

    # Disable demangling since gdb will treat names starting
    # with `'_Z` (The marker for Itanium ABI) specially.
    gdb.execute("set demangle-style none")

    coro_frame_type = gdb.lookup_type(block.function.linkage_name + ".coro_frame_ty")
    coro_frame_ptr_type = coro_frame_type.pointer()
    coro_frame = gdb.Value(addr).cast(coro_frame_ptr_type).dereference()

    gdb.execute("set demangle-style auto")
    gdb.write(coro_frame.format_string(pretty_structs = True))

```

ShowCoroFrame()

Then let's run:

```

1 $ clang++ -std=c++20 -g debugging-example.cpp -o debugging-example
2 $ gdb ./debugging-example
3 (gdb) # We've already set the breakpoint.
4 (gdb) r
5 Program received signal SIGTRAP, Trace/breakpoint trap.
6 detail::chain_fn<0> () at debugging-example2.cpp:73
7 73      co_return 0;
8 (gdb) # Executes the debugging scripts
9 (gdb) source debugging-helper.py
10 (gdb) # Print the asynchronous stack
11 (gdb) async-bt
12 #0 0x401c40 in detail::chain_fn<0>() ['frame_addr = 0x441860', 'promise_addr = 0x441870',
13   'continuation_addr = 0x441870'] at debugging-example.cpp:71
14 #1 0x4022d0 in detail::chain_fn<1>() ['frame_addr = 0x441810', 'promise_addr = 0x441820',
15   'continuation_addr = 0x441820'] at debugging-example.cpp:66
16 #2 0x403060 in detail::chain_fn<2>() ['frame_addr = 0x4417c0', 'promise_addr = 0x4417d0',

```

```
17     'continuation_addr = 0x4417d0'] at debugging-example.cpp:66
18 #3 0x403df0 in detail::chain_fn<3>() ['frame_addr = 0x441770', 'promise_addr = 0x441780',
19     'continuation_addr = 0x441780'] at debugging-example.cpp:66
20 #4 0x404b80 in detail::chain_fn<4>() ['frame_addr = 0x441720', 'promise_addr = 0x441730',
21     'continuation_addr = 0x441730'] at debugging-example.cpp:66
22 #5 0x405910 in detail::chain_fn<5>() ['frame_addr = 0x4416d0', 'promise_addr = 0x4416e0',
23     'continuation_addr = 0x4416e0'] at debugging-example.cpp:66
24 #6 0x4066a0 in detail::chain_fn<6>() ['frame_addr = 0x441680', 'promise_addr = 0x441690',
25     'continuation_addr = 0x441690'] at debugging-example.cpp:66
26 #7 0x407430 in detail::chain_fn<7>() ['frame_addr = 0x441630', 'promise_addr = 0x441640',
27     'continuation_addr = 0x441640'] at debugging-example.cpp:66
28 #8 0x4081c0 in detail::chain_fn<8>() ['frame_addr = 0x4415e0', 'promise_addr = 0x4415f0',
29     'continuation_addr = 0x4415f0'] at debugging-example.cpp:66
30 #9 0x408f50 in detail::chain_fn<9>() ['frame_addr = 0x441590', 'promise_addr = 0x4415a0',
31     'continuation_addr = 0x4415a0'] at debugging-example.cpp:66
32 #10 0x409ce0 in detail::chain_fn<10>() ['frame_addr = 0x441540', 'promise_addr = 0x441550',
33     'continuation_addr = 0x441550'] at debugging-example.cpp:66
34 #11 0x40aa70 in detail::chain_fn<11>() ['frame_addr = 0x4414f0', 'promise_addr = 0x441500',
35     'continuation_addr = 0x441500'] at debugging-example.cpp:66
36 #12 0x40b800 in detail::chain_fn<12>() ['frame_addr = 0x4414a0', 'promise_addr = 0x4414b0',
37     'continuation_addr = 0x4414b0'] at debugging-example.cpp:66
38 #13 0x40c590 in detail::chain_fn<13>() ['frame_addr = 0x441450', 'promise_addr = 0x441460',
39     'continuation_addr = 0x441460'] at debugging-example.cpp:66
40 #14 0x40d320 in detail::chain_fn<14>() ['frame_addr = 0x441400', 'promise_addr = 0x441410',
41     'continuation_addr = 0x441410'] at debugging-example.cpp:66
42 #15 0x40e0b0 in detail::chain_fn<15>() ['frame_addr = 0x4413b0', 'promise_addr = 0x4413c0',
43     'continuation_addr = 0x4413c0'] at debugging-example.cpp:66
44 #16 0x40ee40 in detail::chain_fn<16>() ['frame_addr = 0x441360', 'promise_addr = 0x441370',
45     'continuation_addr = 0x441370'] at debugging-example.cpp:66
46 #17 0x40fb0 in detail::chain_fn<17>() ['frame_addr = 0x441310', 'promise_addr = 0x441320',
47     'continuation_addr = 0x441320'] at debugging-example.cpp:66
48 #18 0x410960 in detail::chain_fn<18>() ['frame_addr = 0x4412c0', 'promise_addr = 0x4412d0',
49     'continuation_addr = 0x4412d0'] at debugging-example.cpp:66
50 #19 0x4116f0 in detail::chain_fn<19>() ['frame_addr = 0x441270', 'promise_addr = 0x441280',
51     'continuation_addr = 0x441280'] at debugging-example.cpp:66
52 #20 0x412480 in detail::chain_fn<20>() ['frame_addr = 0x441220', 'promise_addr = 0x441230',
53     'continuation_addr = 0x441230'] at debugging-example.cpp:66
54 #21 0x413210 in detail::chain_fn<21>() ['frame_addr = 0x4411d0', 'promise_addr = 0x4411e0',
55     'continuation_addr = 0x4411e0'] at debugging-example.cpp:66
56 #22 0x413fa0 in detail::chain_fn<22>() ['frame_addr = 0x441180', 'promise_addr = 0x441190',
57     'continuation_addr = 0x441190'] at debugging-example.cpp:66
58 #23 0x414d30 in detail::chain_fn<23>() ['frame_addr = 0x441130', 'promise_addr = 0x441140',
```

```

59     '|continuation_addr = 0x441140'|] at debugging-example.cpp:66
60 #24 0x415ac0 in detail::chain_fn<24>() ['frame_addr = 0x4410e0', 'promise_addr = 0x4410f0',
61     '|continuation_addr = 0x4410f0'|] at debugging-example.cpp:66
62 #25 0x416850 in detail::chain_fn<25>() ['frame_addr = 0x441090', 'promise_addr = 0x4410a0',
63     '|continuation_addr = 0x4410a0'|] at debugging-example.cpp:66
64 #26 0x4175e0 in detail::chain_fn<26>() ['frame_addr = 0x441040', 'promise_addr = 0x441050',
65     '|continuation_addr = 0x441050'|] at debugging-example.cpp:66
66 #27 0x418370 in detail::chain_fn<27>() ['frame_addr = 0x440ff0', 'promise_addr = 0x441000',
67     '|continuation_addr = 0x441000'|] at debugging-example.cpp:66
68 #28 0x419100 in detail::chain_fn<28>() ['frame_addr = 0x440fa0', 'promise_addr = 0x440fb0',
69     '|continuation_addr = 0x440fb0'|] at debugging-example.cpp:66
70 #29 0x419e90 in detail::chain_fn<29>() ['frame_addr = 0x440f50', 'promise_addr = 0x440f60',
71     '|continuation_addr = 0x440f60'|] at debugging-example.cpp:66
72 #30 0x41ac20 in detail::chain_fn<30>() ['frame_addr = 0x440f00', 'promise_addr = 0x440f10',
73     '|continuation_addr = 0x440f10'|] at debugging-example.cpp:66
74 #31 0x41b9b0 in chain() ['frame_addr = 0x440eb0', 'promise_addr = 0x440ec0',
75     '|continuation_addr = 0x440ec0'|] at debugging-example.cpp:77

```

Now we get the complete asynchronous stack! It is also possible to print other asynchronous stack which doesn't live in the top of the stack. We can make it by passing the address of the corresponding coroutine frame to `async-bt` command.

By the debugging scripts, we can print any coroutine frame too as long as we know the address. For example, we can print the coroutine frame for `detail::chain_fn<18>()` in the above example. From the log record, we know the address of the coroutine frame is `0x4412c0` in the run. Then we can:

```

1 (gdb) show-coroutine-frame 0x4412c0
2 {
3     __resume_fn = 0x410960 <detail::chain_fn<18>()>,
4     __destroy_fn = 0x410d60 <detail::chain_fn<18>()>,
5     __promise = {
6         continuation = {
7             _M_fr_ptr = 0x441270
8         },
9         result = 0
10    },
11    struct_Awaiter_0 = {
12        struct_std___n4861__coroutine_handle_0 = {
13            struct_std___n4861__coroutine_handle = {
14                PointerType = 0x441310
15            }
16        }
17    },
18    struct_task_1 = {
19        struct_std___n4861__coroutine_handle_0 = {

```

```

20     struct_std____n4861__coroutine_handle = {
21         PointerType = 0x0
22     }
23 }
24 },
25 struct_task__promise_type__FinalSuspend_2 = {
26     struct_std____n4861__coroutine_handle = {
27         PointerType = 0x0
28     }
29 },
30     __coro_index = 1 '\001',
31     struct_std____n4861__suspend_always_3 = {
32         __int_8 = 0 '\000'
33 }

```

40.8 Get the living coroutines

Another useful task when debugging coroutines is to enumerate the list of living coroutines, which is often done with threads. While technically possible, this task is not recommended in production code as it is costly at runtime. One such solution is to store the list of currently running coroutines in a collection:

```

1 inline std::unordered_set<void*> lived_coroutines;
2 // For all promise_type we want to record
3 class promise_type {
4 public:
5     promise_type() {
6         // Note to avoid data races
7         lived_coroutines.insert(
8             std::coroutine_handle<promise_type>::from_promise(*this).address());
9     }
10    ~promise_type() {
11        // Note to avoid data races
12        lived_coroutines.erase(
13            std::coroutine_handle<promise_type>::from_promise(*this).address());
14    }
15 };

```

In the above code snippet, we save the address of every lived coroutine in the `lived_coroutines` `unordered_set`. As before, once we know the address of the coroutine we can derive the function, `promise_type`, and other members of the frame. Thus, we could print the list of lived coroutines from that collection.

Please note that the above is expensive from a storage perspective, and requires some level of locking (not pictured) on the collection to prevent data races.

Chapter 41

C++20 Coroutines and io_uring

👤 pablo ariasal 📅 2022-11-13 💬 ★★

41.1 C++20 Coroutines and io_uring - Part 1/3

As part of my job I have to deal with quite heavy I/O loads. Multiple times I've jumped into a fighting ring facing challenges that demand me to perform thousands of I/O operations in the most efficient way possible. And I haven't always won. This has directed my attention to more powerful weapons, weapons like `io_uring`. `io_uring` is a new asynchronous I/O API in the Linux kernel which offers efficiency and scalability never seen before. With coroutines accepted into the C++20 standard and matured enough implementations at my disposal, I had all I need to forge the ultimate weapon, or at the very least an irresistible hobby project.

In this series of posts we'll write a program that reads lots of files stored on disk using Linux `io_uring` and C++20 coroutines. The main motivation comes from the fact that there are lots of in-depth resources for both `io_uring` and C++20 coroutines, but practically nothing showing how to combine both. We will discover that asynchronous I/O and coroutines go as well together as bread and butter.

The series will be split into three parts. In the first part we will solve the problem using `io_uring` only. In the second part we will remodel the implementation on top of C++20 coroutines. In the third part we will optimize the program by making it multi-threaded, this will reveal the true power of coroutines.

41.1.1 Async I/O Coroutines Permalink

Asynchronous I/O, as opposed to synchronous I/O, describes the idea of performing an I/O operation without blocking the calling thread. Instead of waiting until the operation finishes, the thread is released immediately and can do other work while the requested operation is being performed in the background. After a while the calling thread (or some other one) can come back and collect the results of the requested operation. Its like when you tell your pizza guy: "put a margherita in the oven, I'll go grab something from the pharmacy and will be right back".

Asynchronous I/O can be a lot more efficient than synchronous I/O, threads don't need to wait for resources to become available, but at the same time it makes programs more complex. Programs need to work, well, asynchronously, they must remember to pick up the margheritas they ordered.

Coroutines allow us to write asynchronous code as if it was synchronous. If you write a coroutine that loads a file from disk, it can suspend execution and return to the caller while the file is being loaded, and

resume once the data is available. All written as if it was a good-old synchronous call.

Combining asynchronous I/O and coroutines allows us to write asynchronous programs without actually writing asynchronous code. The best of both worlds.

41.1.2 Goals

In this post series we will be writing a program that reads and parses hundreds of wavefront OBJ¹ files from disk.

Its important to understand the distinction between reading and parsing. Reading denotes the action of loading the contents of a file from disk into a memory buffer. Parsing means taking the data in the buffer and transforming into into a data structure that makes sense to the application.

An OBJ is an ASCII file that describes the geometry of one or more 3D triangular meshes. The file encodes information like the triangles that make up the mesh, their vertices, colors, etc. Parsing an OBJ file means converting its ASCII representation into a C++ objects that has a convenient API for accessing the meshes attributes.

For parsing the OBJ files we will use a library called tinyobjloader². It receives a string containing the contents of the OBJ file and parses it into a ObjReader object:

```
1 std::string obj_data = // the contents of the OBJ file
2 tinyobj::ObjReader reader;
3 reader.ParseFromString(obj_data);
```

We can use the API of reader to access the attributes of the shapes defined in the file. For example, to list how many shapes are defined in the file you can do the following:

```
1 std::cout << reader.GetShapes().size() << '\n';
```

41.1.3 Ground Work

First of all let's lay down some ground work by defining some basic abstractions that will make our implementation easier and safer.

We will be dealing with files, so we write an RAII class that manages a read-only file:

```
1 class ReadOnlyFile {
2 public:
3     ReadOnlyFile(const std::string &file_path) : path_{file_path} {
4         fd_ = open(file_path.c_str(), O_RDONLY);
5         if (fd_ < 0) {
6             throw std::runtime_error("Fail to open file");
7         }
8         size_ = get_file_size(fd_);
9         if (size_ < 0) {
10             throw std::runtime_error("Fail to get size of file");
11     }
```

¹https://en.wikipedia.org/wiki/Wavefront_.obj_file

²<https://github.com/tinyobjloader/tinyobjloader>

```

12     }
13
14     ReadOnlyFile(ReadOnlyFile &&other)
15         : path_{std::exchange(other.path_, {})},
16           fd_{std::exchange(other.fd_, -1)},
17           size_{other.size()} {}
18
19     ~ReadOnlyFile() {
20         if (fd_) {
21             close(fd_);
22         }
23     }
24
25     int fd() const { return fd_; }
26     off_t size() const { return get_file_size(fd_); }
27     const std::string &path() const { return path_; }
28
29 private:
30     std::string path_;
31     int fd_;
32     off_t size_;
33 };

```

Quite simple, it opens a file in read only mode and closes it on destruction. We do some (very clumsy) error handling and implement a move constructor, so that the type can be stored inside some STL containers like

```
1 std::vector.
```

Another type we will use a lot is Result:

```

1 struct Result {
2     tinyobj::ObjReader result; // stores the actual parsed obj
3     int status_code{0};       // the status code of the read operation
4     std::string file;        // the file the OBJ was loaded from
5 };

```

This is the final result of parsing of an OBJ file, it contains the final OBJ object, the corresponding file path, as well as the status code of the read call.

The goal of our program is to load a list of OBJ files and return a `std::vector<Result>`.

41.1.4 First Attempt: A Trivial Approach

Now that we have laid down some ground work we can continue with the main dish. As usual, let's start with the easiest possible implementation: a single-threaded blocking file loader.

```

1 Result readSynchronous(const ReadOnlyFile &file) {
2     Result result{.file = file.path()};

```



```

9     })
10    .wait();
11    return result;
12 }

```

Here we are using bshoshany's thread-pool library³ to run individual iterations of the for-loop in parallel. Every thread receives a fixed number of iterations, denoted by the range [a,b). You can also use something like openMP, or `std::async`, the idea is the same.

This is better, even when the threads are still being blocked on read and a system call is performed per file, multiple files are being processed in parallel. The overhead in terms of code changes is very small, and it opens further optimization opportunities: we can for example allocate a single buffer per thread that can be reused by multiple files.

For many applications this approach is more than sufficient, but imagine that you developing a web server, listening to thousands of socket connections at once. Would you create a thread per socket? Probably not. What you need is a way of telling the OS: "Look, I'm interested in these sockets, let me know when any of them has any data to be read, I'll continue with my day". What you need is asynchronous I/O.

41.1.6 Enter io_uring

`io_uring` is a new asynchronous I/O API available in the Linux kernel (version 5.1 upwards). This is not the first API of its kind, there are other options available in the kernel, like `epoll`, `poll`, `select` and `aio`, all with their issues and limitations. `io_uring` aims to start a new page and become the standard API for all asynchronous I/O operations in the kernel.

The API is called `io_uring` because its based on two ring buffers: the submission queue and the completion queue. The buffers are shared between user code and the kernel and writing to or reading from them does not require system calls or copies.

The idea is simple: user code writes requests into the submission queue and submits them to the kernel, the kernel consumes the requests in the queue, performs the requested operations, and writes the results into the completion queue. User code can asynchronously retrieve the completed requests from the queue at a later point.

The raw API of `io_uring` is a little complex, this is why programs usually use a wrapper library called liburing (created by Jens Axboe, original author of `io_uring`), which extracts most of the boilerplate away and provides convenient utilities for using `io_uring`.

41.1.7 Parsing OBJS with liburing

We will now implement our program using `liburing`.

First of all we define a thin wrapper RAII class that initializes an `io_uring` object and frees it upon destruction:

```

1 class IOUring {
2 public:
3     explicit IOUring(size_t queue_size) {
4         if (auto s = io_uring_queue_init(queue_size, &ring_, 0); s < 0) {

```

³<https://github.com/bshoshany/thread-pool>

```

5         throw std::runtime_error("error initializing io_uring: " + std::to_string(s));
6     }
7 }
8
9 IOUring(const IOUring &) = delete;
10 IOUring &operator=(const IOUring &) = delete;
11 IOUring(IOUring &&) = delete;
12 IOUring &operator=(IOUring &&) = delete;
13
14 ~IOUring() { io_uring_queue_exit(&ring_); }
15
16 struct io_uring *get() {
17     return &ring_;
18 }
19
20 private:
21     struct io_uring ring_;
22 };

```

`io_uring_queue_init` initializes an instance of `io_uring` with a given queue size (this is the size of the ring buffers of the completion and submission queues). `io_uring_queue_exit` destroys an `io_uring` instance.

Let's now try to implement our OBJ loader using liburing.

The implementation must consist of two parts: first we must push the file read requests to the submission queue. Second, we have to wait for completion requests to arrive at the completion queue and parse the contents of the buffer as they come.

```

1 std::vector<Result> iouringOBJLoader(const std::vector<ReadOnlyFile> &files) {
2     IOUring uring{files.size()};
3     auto buffs = initializeBuffers(files);
4     pushEntriesToSubmissionQueue(files, buffs, uring);
5     return readEntriesFromCompletionQueue(files, buffs, uring);
6 }

```

We start by creating an `io_uring` instance with enough queue capacity to hold an entry for each file. Next we allocate the buffers, one per file.

`pushEntriesToSubmissionQueue` writes the submission entries into the submission queue:

```

1 void pushEntriesToSubmissionQueue(const std::vector<ReadOnlyFile> &files,
2                                     const std::vector<std::vector<char>> &buffs,
3                                     IOUring &uring) {
4     for (size_t i = 0; i < files.size(); ++i) {
5         struct io_uring_sqe *sqe = io_uring_get_sqe(uring.get());
6         io_uring_prep_read(sqe, files[i].fd(), buffs[i].data(), buffs[i].size(), 0);
7         io_uring_sqe_set_data64(sqe, i);
8     }
9 }

```

`io_uring_get_sqe` creates an entry in the submission queue, `sqe`. We can now set the contents of the submission entry with `io_uring_prep_read`, which specifies that we want the kernel to read the contents of the file pointed to by `files[i].fd()` into the buffer `buffs[i]`.

One can also append user data to the submission entry with `io_uring_sqe_set_data`. This is data that is not used by the kernel but is just copied into the corresponding completion entry of the current request. This is important in order to differentiate which completion entry corresponds to which submission entry. In this case we just write the index of the file, which we can use to reference back when the read operation completes.

After we have written all the submission entries into the queue its time to submit them to the kernel and wait until they start appearing in the completion queue. Once completion entries appear, we can read the corresponding buffers into OBJ files:

```

1 std::vector<Result> readEntriesFromCompletionQueue(const std::vector<ReadOnlyFile> &files,
2                                                 const std::vector<std::vector<char>> &buffs,
3                                                 IOUring &uring) {
4     std::vector<Result> results;
5     results.reserve(files.size());
6
7     while (results.size() < files.size()) {
8         io_uring_submit_and_wait(uring.get(), 1);
9         io_uring_cqe *cqe;
10        unsigned head;
11        int processed{0};
12        io_uring_for_each_cqe(uring.get(), head, cqe) {
13            auto id = io_uring_cqe_get_data64(cqe);
14            results.push_back({.status_code = cqe->res, .file = files[id].path()});
15            if (results.back().status_code) {
16                readObjFromBuffer(buff[0], results.back().result);
17            }
18            ++processed;
19        }
20
21        io_uring_cq_advance(uring.get(), processed);
22    }
23
24    return results;
}
```

First we call `io_uring_submit_and_wait`, which submits all entries to the kernel and blocks until at least one completion request has been pushed into the completion queue.

Once we have completion entries in the queue we can start processing them. `io_uring_for_each_cqe` is a macro defined in `liburing` whose body gets executed for every completion entry in the completion queue.

This is what we must do after a completion entry arrives:

- get the id of the file this completion entry corresponds to. This is the same id we wrote into the corresponding submission entry.

- Write the status code into the `Result` object. This is the status code of the `read` operation performed by the kernel.
- If the read was successful parse the OBJ from the buffer into the `Result` object.

Finally we can liberate some space in the submission queue as we have processed some completion entries. For this we use `io_uring_cq_advance`, which does nothing more than moving head of the ring buffer back by n elements, making room for more entries.

41.1.8 Closing Thoughts

The big advantage of the `io_uring` implementation is the substantial reduction of syscalls performed. In fact, using strace we can see that the `io_uring` implementation has 512 less system calls than in the synchronous case, this is due to the `read` syscalls:

```

1 > strace -c -e read -- ./build_release/couring --trivial
2 Running trivial
3 Processed 512 files.
4 % time      seconds    usecs/call      calls      errors syscall
5 -----
6 100.00      0.000591           1        517          0   read
7
8 > strace -c -e read -- ./build_release/couring --iouring
9 Running iouring
10 Processed 512 files.
11 % time      seconds    usecs/call      calls      errors syscall
12 -----
13 100.00      0.000053           10         5          0   read

```

Instead of performing one read syscall per file, we perform none. Note that reading and writing to `io_uring` queues is not a syscall and doesn't involve a context switch.

Our `io_uring` implementaion approach still has some issues, though. First, the code is substantially more complex than the synchronous case, we no longer have a single synchronous function that reads and parses an OBJ file, but rather a program that writes to a queue and then polls it. This makes is very hard to extend.

Second, our program is CPU-bound. As some profiling shows, most of the time is spent parsing the OBJs:

```

1      %  cumulative   self
2  time  seconds   seconds
3  31.25      0.05     0.05  tinyobj::tryParseDouble(char const*, char const*, double*)
4  18.75      0.08     0.03  tinyobj::LoadObj(tinyobj::attrib_t*, std::vector<tinyobj::shape_t,
5  12.50      0.10     0.02  allDone(std::vector<Task, std::allocator<Task> > const&)
6  12.50      0.12     0.02  tinyobj::parseReal(char const**, double)
7  6.25       0.13     0.01  tinyobj::parseReal3(float*, float*, float*, char const**, double, do
8 ...

```

The parsing of the OBJ files still happens on a single thread. While processing the submission entries happens

on a thread pool inside the kernel, consuming the completion entries happens in a single thread in user space. Is clear that we must parallelize the parsing path.

In the next parts of the series we will try to solve these problems.

41.2 C++20 Coroutines and io_uring - Part 2/3

In the first part of the series we learned about `io_uring` by writing a program that reads and parses hundreds of OBJ files from disk. In this second part of the series we will rewrite that program of by making use of C++20 coroutines.

Even though required basic concepts are explained, this post is not meant to be an in-depth dive into C++ coroutines, the goal is rather to showcase of how coroutines can be used together with `io_uring`. If you are not familiar with C++ coroutines yet, Lewis Baker's blog⁴ series is a great place to start.

41.2.1 Basic Idea

What we want to do is simple: implement a coroutine that loads and parses an OBJ file from disk using `io_uring`. When this coroutine is called it pushes a submission request into the submission queue and suspends execution, returning to the caller. Once the corresponding completion entry has arrived the coroutine is resumed and the OBJ can be parsed:

```

1 Task parseOBJFile(IOUring &uring, const ReadOnlyFile &file) {
2     std::vector<char> buff(file.size());
3     int status = co_await ReadFileAwaitable{uring, file, buff};
4     // completion entry has arrived at this point
5     // buffer has been filled and parsing can take place
6     Result result{.status_code = 0, .file = file.path()};
7     readObjFromBuffer(buff, result.result);
8     co_return result;
9 }
```

This is our coroutine. It looks similar to a normal function, in fact, it's just good old imperative synchronous code: allocate a buffer, read contents of file into buffer, parse contents of buffer, return result.

The coroutine is naturally not executed synchronously: it suspends execution and yields control to the caller at predetermined locations. Most notably, at the await expression in the second line: `co_await ReadFileAwaitable uring, file, buff`. This is the point where the submission entry is pushed to the submission queue and control returns to the caller. `co_await` is an operator that is applied to an awaitable and suspends execution. `ReadFileAwaitable` is such an awaitable, its task is to suspend the coroutine and register it to be woken up in the future. Once the coroutine is resumed execution continues right after the `co_await` expression.

41.2.2 Suspending Execution

`ReadFileAwaitable` is the awaitable we use used to suspend the coroutine. Don't be scared we will explain what's going on here:

⁴<https://app.yinxiang.com/shard/s9/nl/16427418/3abecf16-ca96-42c7-be8b-87892b6bcf9e>

```

1  class ReadFileAwaitable {
2  public:
3      ReadFileAwaitable(IOUring &uring, const ReadOnlyFile &file,
4                         const std::vector<char> &buf) {
5          sqe_ = io_uring_get_sqe(uring.get());
6          io_uring_prep_read(sqe_, file.fd(), buf.data(), buf.size(), 0);
7      }
8
9      auto operator co_await() {
10         struct Awariter {
11             io_uring_sqe *entry;
12             RequestData requestData;
13
14             Awaitable(io_uring_sqe *sqe) : entry{sqe} {}
15             bool await_ready() { return false; }
16             void await_suspend(std::coroutine_handle<> handle) noexcept {
17                 requestData.handle = handle;
18                 io_uring_sqe_set_data(entry, &requestData);
19             }
20             int await_resume() { return requestData.statusCode; }
21         };
22
23         return Awariter{sqe_};
24     }
25
26 private:
27     io_uring_sqe *sqe_;
28 };

```

`ReadFileAwaitable` creates a submission queue entry and pushes it into the submission queue. This makes sense, as we must push the request into the queue before suspending the coroutine.

`ReadFileAwaitable` has two member variables: `entry` and `requestData`. `entry` is a pointer to the submission entry, we need it to be able to attach user data to it with `io_uring_sqe_set_data`. User data is arbitrary data (literally a `void*`) that we can attach to a submission request. The user data is propagated by the kernel as-is to the corresponding completion entry and its usually used to uniquely distinguish a completion entry and link it to the corresponding submission request.

`ReadFileAwaitable` overloads operator `co_await()`, which returns an awariter: `Awariter`. `Awariter` implements the required member functions dictated by the standard: `await_ready`, `await_suspend` and `await_resume`.

`await_ready` returns always false. This means that the coroutine will always be suspended on `co_await` `ReadFileAwaitable`. We could have also inherited from `std::suspend_always`.

Now we must implement `await_suspend`. This function is called right after the coroutine has been suspended and receives the coroutine handle as parameter. This is a great place to write the user data to the

submission queue entry. But what should we write as user data?

Let's think for minute: we pushed a submission entry into the submission queue. There will be a corresponding completion entry arriving soon at the completion queue. Once the completion entry has arrived we must resume the coroutine that was just suspended. What do we need to resume the coroutine? The coroutine handle that has just been passed to us! Let's write the coroutine handle as user data into the submission request!

The coroutine handle is stored inside a member variable of `Awaiter,requestData` of type `requestData`:

```

1 struct requestData {
2     std::coroutine_handle<> handle;
3     int statusCode{-1};
4 };

```

`requestData` stores the coroutine handle and the status code of the read operation. The status code will be written to the `Awaiter` object later on, when the completion request has arrived. The user data is therefore a pointer to the `requestData` data member of the `awaiter` object.

Finally we can implement `await_resume`. `await_resume` is called right before the coroutine is resumed and returns the status code of the read operation. In other words, we can assume that `requestData.statusCode` has been initialized by the time `await_resume` is called.

The return value of `await_resume` is the result of the entire `await` expression, meaning that we can write:

```

1 int status = co_await ReadFileAwaitable{}
2 if (!status) {
3     // ...
4 }

```

as if it were a normal read call.

41.2.3 Resuming Execution

Calling `co_await` `ReadFileAwaitable` puts the coroutine to sleep, but how do we wake it up? Easy, we wait for completion requests to arrive at the completion queue and after a completion entry has arrived we extract the coroutine handle inside it and use it to resume the coroutine:

```

1 int consumeCQEntries(IOUring &uring) {
2     int processed{0};
3     io_uring_cqe *cqe;
4     unsigned head; // head of the ring buffer, unused
5     io_uring_for_each_cqe(uring.get(), head, cqe) {
6         auto *request_data = static_cast<requestData *>(io_uring_cqe_get_data(cqe));
7         // make sure to set the status code before resuming the coroutine
8         request_data->statusCode = cqe->res;
9         request_data->handle.resume(); // await_resume is called here
10        ++processed;
11    }
12    io_uring_cq_advance(uring.get(), processed);

```

```

13     return processed;
14 }
```

Before resuming the coroutine we have to write the status code into `request_data`. `request_data` is a pointer to the `requestData` data member in the `Awaiter` object that is returned in `await_resume`.

We write now a `consumeCQEEntriesBlocking` helper function which submits pending submission entries to the kernel and blocks until at least one completion entry arrives:

```

1 int consumeCQEEntriesBlocking(IOUring &uring) {
2     io_uring_submit_and_wait(uring.get(), 1); // blocks if queue empty
3     return consumeCQEEntries(uring);
4 }
```

We have learned the mechanics of suspending and resuming our coroutine, now we can write the client code that uses it to load the OBJ files.

Intuitively we must allocate some `std::vector` that contains the results returned by `parseOBJFile`, but what is the return type of `parseOBJFile`? What is the return type of a coroutine? It's a coroutine type called `Task`.

41.2.4 Coroutine Type: Task

`Task` is the return type of our coroutine. We must implement it ourselves by defining the API demanded by the standard. Again, don't worry we will go through it:

```

1 class Task {
2 public:
3     struct promise_type {
4         Result result;
5
6         Task get_return_object() { return Task(this); }
7
8         void unhandled_exception() noexcept {}
9
10        void return_value(Result result) noexcept { result = std::move(result); }
11        std::suspend_never initial_suspend() noexcept { return {}; }
12        std::suspend_always final_suspend() noexcept { return {}; }
13    };
14
15    explicit Task(promise_type *promise)
16        : handle_{HandleT::from_promise(*promise)} {}
17    Task(Task &&other) : handle_{std::exchange(other.handle_, nullptr)} {}
18
19    ~Task() {
20        if (handle_) {
21            handle_.destroy();
22        }
23    }
24}
```

```

23     }
24
25     Result getResult() const & {
26         assert(handle_.done());
27         return handle_.promise().result;
28     }
29
30     Result&& getResult() && {
31         assert(handle_.done());
32         return std::move(handle_.promise().result);
33     }
34
35     bool done() const { return handle_.done(); }
36
37     using HandleT = std::coroutine_handle<promise_type>;
38     HandleT handle_;
39 };
40

```

Task defines a promise type. The promise object is instantiated for every call to the coroutine and lives inside the coroutine frame. The promise object is used to transmit the result of the coroutine (or an exception if one is thrown inside the coroutine body). Hence, the promise type has a member `Result` result, which contains the final result of the parsing of an OBJ file:

```

1 struct Result {
2     tinyobj::ObjReader result; // stores the actual parsed obj
3     int status_code{0};        // the status code of the read operation
4     std::string file;         // the file the OBJ was loaded from
5 };

```

`result` is initialized from inside the member function `return_value`, which is called when the coroutine reaches an `co_return` statement.

Task defines a convenience member function `getResult()` that extracts and returns the result from the promise.

`promise_type` must define a member function `get_return_object` which returns the actual coroutine object. In our case its an instance of Task. `unhandled_exception` is called when the coroutine body throws, we left it unimplemented as we are exception free (or aim to be). `initial_suspend` and `final_suspend` determine the initial and final suspension behavior of the coroutine, whether the coroutine should start or finish in a suspended state or not.

Task contains the coroutine handle `handle_` and manages its lifetime: it destroys the coroutine frame in upon destruction by calling `handle_.destroy()`. It also defines a `done()` convenience member that tells whether the coroutine has finished executing.

C++20 coroutines are raw, this means that instead of being a finished cake, they are more like a bunch of flour, eggs and butter. In order to implement a coroutine you have to write a lot of supporting code and boilerplate, you must bake your own cake. For this reason it is usual to make use of a coroutine library like

cppcoro by Lewis Baker⁵, which implement generic version of our Task type, among a lot of more useful abstractions, greatly reducing the amount of boilerplate.

41.2.5 Putting it all together

Now we can implement the top-level function that parses the OBJ files using coroutines:

```

1 std::vector<Result> parseOBJFiles(const std::vector<ReadOnlyFile> &files) {
2     IOUring uring{files.size()};
3     std::vector<Task> tasks;
4     tasks.reserve(files.size());
5     for (const auto &file : files) {
6         tasks.push_back(parseOBJFile(uring, file));
7     }
8     while (!allDone(tasks)) {
9         // consume all entries in the submission queue
10        // if the queue is empty block until the next completion arrives
11        consumeCQEEntriesBlocking(uring);
12    }
13    return gatherResults(tasks);
14 }
```

It allocates a vector of Tasks by executing the `parseOBJFile` coroutine for each file. Note that `initial_suspend` in `promise_type` returns `std::suspend_never`, this means that the `parseOBJFile` coroutine doesn't start suspended but right away continues execution until `co_await ReadFileAwaitable`, where the coroutine is suspended.

Once all coroutines are suspended and the kernel is doing its work, we don't have other option other than wait until some completion requests start appearing in the completion queue, `consumeCQEEntriesBlocking` wakes up the coroutines one-by-one as the corresponding completion entries start arriving.

`allDone` is a simple helper that checks if all coroutines have been fully executed to completion:

```

1 bool allDone(const std::vector<Task> &tasks) {
2     return std::all_of(tasks.cbegin(), tasks.cend(),
3                        [](const auto &t) { return t.done(); });
4 }
```

Upon resumption coroutines parse the OBJ file and return the result with `co_return` result;

Finally we can extract the final results from the finished coroutines:

```

1 std::vector<Result> gatherResults(const std::vector<Task> &tasks) {
2     std::vector<Result> results;
3     results.reserve(tasks.size());
4     for (auto &&t : tasks) {
5         results.push_back(std::move(t).getResult());
6     }

```

⁵<https://github.com/lewissbaker/cppcoro>

```

7     return results;
8 }
```

At the end of the function's block scope, all Tasks are destroyed, deallocating all coroutine frames.

41.2.6 Closing Thoughts

You may be asking yourself: what is the actual point of all this compared to the implementation without coroutines? It's a fair question. One may argue that we have just added unnecessary boilerplate to an already straight-forward implementation. It's also not particularly more efficient: we are still performing parsing sequentially in a single thread.

Luckily we are not done yet. See, the beauty of coroutines is that they are very composable. Once you have a coroutine-based implementation adding more awaitables or other coroutines is child's play, they fit like lego stones.

In part 3 we will extend our coroutine-based implementation such that parsing is performed in parallel in a thread pool. This is where the real power of coroutines will come to light.

41.3 C++20 Coroutines and io_uring - Part 3/3

You have made it to the last part of the series. In part 2 we wrote a coroutine-based program that loads and parses a list of OBJ files using coroutines and `io_uring`. The program still has a big disadvantage: it's CPU-bound. Parsing of OBJ files, the most costly part of the algorithm, is performed sequentially on a single thread.

The root of the problem is that our coroutines are resumed on the main thread. Ideally we would like to wake them up in different threads, so that multiple parsings can be conducted in parallel.

This is exactly what we will do. We will add a second await expression inside our coroutine: `co_await pool.schedule()`. This will cause our coroutine to be suspended and scheduled to be resumed in a thread pool:

```

1 Task parseOBJFile(IOUring &uring, const ReadOnlyFile &file, ThreadPool &pool) {
2     std::vector<char> buff(file.size());
3     int status = co_await ReadFileAwaitable(uring, file, buff);
4     co_await pool.schedule();
5     // This is now running on a worker thread
6     Result result{.status_code = 0, .file = file.path()};
7     readObjFromBuffer(buff, result.result);
8     co_return result;
9 }
```

41.3.1 ThreadPool

`ThreadPool` builds the core of our multi-threaded implementation.

`ThreadPool` wraps around a bshoshany Thread Pool's object.⁶ Its API is very simple, you use `push_task` to schedule tasks to be run on the thread pool.

⁶<https://github.com/bshoshany/thread-pool>

```

1  class ThreadPool {
2  public:
3      auto schedule() {
4          struct Awaite : std::suspend_always {
5              BS::thread_pool &tpool;
6              Awaitable(BS::thread_pool &pool) : tpool{pool} {}
7              void await_suspend(std::coroutine_handle<> handle) {
8                  tpool.push_task([handle, this]() { handle.resume(); });
9              }
10         };
11         return Awaite{pool_};
12     }
13
14     size_t numUnfinishedTasks() const { return pool_.get_tasks_total(); }
15
16 private:
17     BS::thread_pool pool_;
18 };

```

ThreadPool defines a member function `schedule()` which returns an instance of Awaite. When a coroutine `co_awaits` the Awaite object it's immediately suspended (note that Awaite inherits from `std::suspend_always`) and schedules it to be resumed in a worker thread in `await`.

This is beautiful! By simply writing `co_await pool.schedule()` we can wake up the current coroutine in a different thread, greatly improving the performance of our application. Parsing of OBJ files happens in parallel.

41.3.2 Multi-Threaded Implementation

That's it, now we can write the top-level function that uses our new coroutine to load and parse the OBJ files:

```

1  std::vector<Result> coroutinesThreadPool(const std::vector<ReadOnlyFile> &files) {
2      IOUring uring{files.size()};
3      ThreadPool pool;
4      std::vector<Task> tasks;
5      tasks.reserve(files.size());
6      for (const auto &file : files) {
7          tasks.push_back(parseOBJFile(uring, file, pool));
8      }
9      io_uring_submit(uring.get());
10     while (pool.numUnfinishedTasks() > 0 || !allDone(tasks)) {
11         // consume entries in the completion queue
12         // return immediately if the queue is empty
13         consumeCQEntriesNonBlocking(uring);

```

```

14     }
15
16     return gatherResults(tasks);
17 }
```

We initialize a thread pool and `io_uring` instance. We call `parseOBJFile` for each file, which will fill up the submission queue. Later we submit the requests to the kernel using `io_uring_submit()`.

Once the kernel is reading the files in the background, coroutines are waken up as the corresponding completion entries arrive. This happens in `consumeCQEntriesNonBlocking`:

```

1 int consumeCQEntriesNonBlocking(IOUring &uring) {
2     io_uring_cqe *temp;
3     if (io_uring_peek_cqe(uring.get(), &temp) == 0) {
4         return consumeCQEntries(uring);
5     }
6     return 0;
7 }
```

Which uses `io_uring_peek_cqe` to peek for existing entries in the completion queue and exits immediately if its empty.

We continue to wait until all coroutine have finalized. Since `allDone` is a linear check, we include an early-exit to avoid calling it that often: `pool.numUnfinishedTasks() > 0`. If there are unfinished tasks scheduled in the thread pool its obvious that we are not done yet.

41.3.3 Closing Words

We have made it! Hopefully by now you can appreciate how cool coroutines are, once you have an existing coroutine, adding extra `co_awaits` is child's play. You can find the entire code for the blog series here.⁷

⁷<https://github.com/pabloariasal/couring>

Chapter 42

Instruction-level parallelism in practice: speeding up memory-bound programs with low ILP

✿ Johnny's Software Lab LLC 📅 2022-06-19 💬 ★★★★

Memory-bound problems happen often in working with large classes, pointer chasing, working with trees, hash maps or linked lists. In this post we will talk about instruction-level parallelism: what instruction-level parallelism is, why is it important for your code's performance and how you can add instruction-level parallelism to improve the performance of your memory-bound program.

42.1 A Quick Introduction to Instruction-Level Parallelism

Modern CPUs execute instructions out of order: if the CPU cannot execute the current instruction, it will move on to executing the instructions that come after it and execute them, if possible. Take the following example written in a pseudo-assembly:

```
1 r[i] = a[i] + b[i] + c[i];  
2  
3 register a_val = load(a + i);  
4 register b_val = load(b + i);  
5 register c_val = load(c + i);  
6 register r_val = a_val + b_val;  
7 r_val = r + c_val  
8 store(r_val, r + i);
```

This is the pseudo-assembly for expression $r[i] = a[i] + b[i] + c[i]$. As you can see, there are three load instructions (lines 3, 4 and 5), two additions (line 6 and 7) and one store (line 8). If for some reason, the CPU cannot execute instruction $\text{load}(a + i)$, it can move to execute the next two instructions $\text{load}(b + i)$ and $\text{load}(c + i)$. All three loads are independent of one another and the CPU can execute them independently, even if one of them takes time to execute.

Instruction `a_val + b_val`, however, cannot start executing before `load(a + i)` and `load(b + i)` are done. We say that instruction `a_val + b_val` depends on these two instructions. If any of these two loads is stuck (e.g. waiting for the data from the memory), so will all the instructions that depend on it.

The ability of the CPU to execute instructions out-of-order and to find instructions that are not blocked by the previous instructions and execute them in parallel is called *instruction-level parallelism* (abbreviated ILP). Codes with little dependencies are said to have high ILP. Conversely, codes with long instruction dependency chains have low ILP.

In the context of memory-bound problems, ILP is important. If a load instruction is stuck waiting for the data from the memory, the CPU can “hide” the memory latency by running other instructions that come after it. However, in the presence of a long cross-iteration dependency chain¹, the CPU cannot find useful work and it sits idle.

In the context of memory-bound problems, codes with low ILP are typically codes where there is a chain of memory loads, i.e. the memory load in the current loop iteration depends on the memory load in the previous iteration, etc.

42.2 Analyzing available instruction-level parallelism

In order to increase available ILP, one needs to measure the available ILP. Analyzing instruction-level parallelism is typically done on the instruction level, but this analysis is too low-level and probably not very useful for an average C/C++ developer. As far as I know, there are no tools that are able to automatically tell you if your program is only memory-bound, or it is memory-bound AND exhibits low ILP². In this section we will introduce a type of analysis “by inspection” that can help you estimate how much available ILP your program exhibits.

42.2.1 Code Example with High ILP

To illustrate the analysis, let’s use a few examples. Here is the first example:

```
1 for (int i = 0; i < n; i++) {
2     c[i] = a[i] + b[i];
3 }
```

For ILP analysis, it is very useful to unroll a loop two times, because often there are self-dependencies, i.e. cases where the instruction depends on itself. This is easier to see when the loop is unrolled. Let’s rewrite the loop in some kind of pseudo-assembly.

```
1 for (int i = 0; i < n; i+=2) {
2     a_val_1 = load(a + i);
3     b_val_1 = load(b + i);
4     c_val_1 = a_val_1 + b_val_1;
5     store(c + i, c_val_1);
6 }
```

¹Cross-iteration dependency chain is a dependency chain where the instructions in the current iteration depend on the results from instructions in the previous iteration.

²There is a tool called `llvm-mca`(<https://llvm.org/docs/CommandGuide/llvm-mca.html>) that you can use to perform dependency analysis, but it works only on assembly code

```

7     a_val_2 = load(a + i + 1);
8     b_val_2 = load(b + i + 1);
9     c_val_2 = a_val_2 + b_val_2;
10    store(c + i + 1, c_val_2);
11 }

```

In each iteration there are four pseudo-instructions: two loads, one addition and one store. If you look at the dependencies, instructions `load(a + i)` and `load(b + i)` can be done independently. Instruction `a_val + b_val` depends on the two previous loads so it needs to wait for them to complete. Instruction `store(c + i, c_val)` depends on the instruction `a_val + b_val`. So there is a dependency chain.

But if we extend the analysis across multiple iterations, then it looks a bit different. There are no loop-carried dependencies in this code, i.e. the case where the data needed for the current iteration is calculated in the previous iteration. In other words, the iterations are completely independent of one another. If the CPU is processing iteration where $i = X$ and is stuck waiting for two loads to complete, it can start executing loads from iteration where $i = X + 1$, $i = X + 2$ etc. A code without loop-carried dependencies exhibits relatively high ILP.

42.2.2 Code Example with Low ILP, but no Dependency Chains in Memory Loading

Here is the second example:

```

1 sum = 0;
2 for (int i = 0; i < n; i++) {
3     sum += a[i];
4 }

```

To illustrate what happens, here is the pseudo-assembly of this loop but the loop is unrolled by a factor of two:

```

1 register sum = 0;
2 for (int i = 0; i < n; i+=2) {
3     register a_val_1 = load(a + i);
4     sum = sum + a_val_1;
5     register a_val_2 = load(a + i + 1);
6     sum = sum + a_val_2;
7 }

```

If we perform the dependency analysis, we see that two loads can be done independently. But the instruction `sum + a_val_2` depends on `sum + a_val_1`. So in this case there is a loop-carried dependency. However, load instructions do not have a loop carried dependency, so the CPU can execute them in advance. This is good for us. Load instructions will in the best case have a latency of 3 cycles, whereas addition will have a latency of 1 cycle. So, out-of-order execution will obtain data many instructions in advance, and as soon as the data becomes available it will be appended to the `sum` register.

This example has a lower ILP than the first, because of the loop-carried dependencies, but load instructions do not have a loop-carried dependency so the CPU can execute them out-of-order.

42.2.3 Code Example with Low ILP and Dependency Chains on Memory Loading

Let's look at a code summing up all values in a linked list. Here is the code snippet:

```

1  sum = 0;
2  while (current != 0) {
3      sum += current->val;
4      current = current->next;
5 }
```

Again, here is the pseudo-assembly of the loop unrolled by a factor of two.

```

register sum = 0;
LOOP_START:
(1) if (current == 0) goto LOOP_END;
(2) register val_0 = load(current + OFFSET(val));
(3) sum = sum + val_0;
(4) current = load(current + OFFSET(next));

(5) if (current == 0) goto LOOP_END;
(6) register val_1 = load(current + OFFSET(val));
(7) sum = sum + val_1;
(8) current = load(current + OFFSET(next));

(9) goto LOOP_START;
LOOP_END:
```

To make the analysis easier, below is the dependency graph for a few first iterations of the above loop using *single-assignment form* to make dependencies easier to track (fig. 42.1):³

The question is this: can the CPU execute loads independently? If you look at the dependency graph, the answer is no. There is a very hard dependency chain: if a single load in the left column gets stuck (e.g. because of a cache miss), then all the instructions in all subsequent rows are stuck waiting for this load.

In other words, there is a dependency chain of load instructions, where, to process iteration X + 1, the CPU must have finished processing at least the important part of iteration X. In the cases where there is a high data cache miss rate, long load dependency chains can be a disaster for software performance.

42.3 What Are the Codes with Low ILP?

Under the assumptions that loads are the critical instructions, we know that codes with loop-carried dependencies where a load in iteration $i = X$ depends on the same or another load from the previous iteration $i = X - 1$ are codes with low ILP. But, typically, what are those codes? Here are a few examples:

- Operations on linked lists
- Operations on trees

³Please note that instructions if (...) goto .. are not hard dependencies due to speculation, at least when they are speculated correctly (which is the case here).

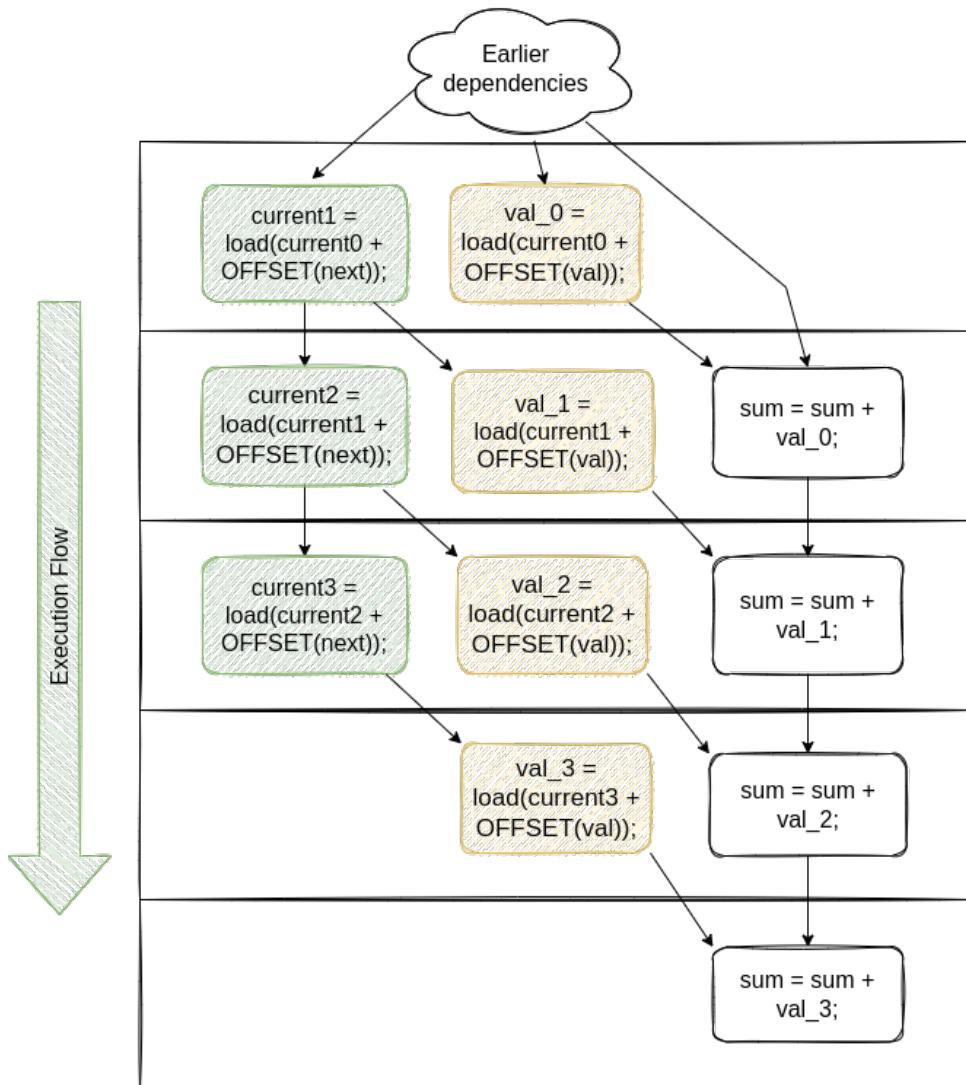


图 42.1: Dependency graph for summing up all values in a linked list

- Operations on hash maps with *separate chaining* used for collision resolution, in cases where the hash map has many conflicts (because it internally uses a linked list).

The reason for low ILP is quite simple. We cannot move to process the next element in the data structure until we have finished processing the current element.

42.4 Techniques Used to Increase Available Instruction-Level Parallelism

In order to add additional ILP to an ILP low memory-bound code, there are two groups of techniques:

- Breaking the dependency chains, where you break the chains by rewriting your datastructure and algorithm.
- Interleaving additional work, where you don't change your datastructure but process more than one element at a time.

The second technique is more useful because it doesn't require rewriting the whole datastructure. The first requires more thinking, and by breaking dependency chains you lose some flexibility of pointers.

To my knowledge, there is no extensive list of techniques used to increase available instruction-level parallelism. Here we will present a few ideas on how to increase the available instruction-level parallelism for memory-bound code, but under no circumstances does this list presents all that is possible to do.

42.4.1 Interleaving Additional Work

Consider the task of looking up N values in a tree or a linked list. Instead of traversing the data structure individually, you can traverse the data structure for all N values at once. Here is how it is done.

42.4.1.1 Linked Lists

When looking up N values in a linked list, instead of traversing the linked list N times, you traverse the linked list only once. While visiting node X , you check if it matches each of N values you want to look up. Then you move to the next node and repeat the process, etc. Here is the code snippet:

```

1 current_node = head;
2
3 while (current_node) {
4     for (int i = 0; i < N; i++) {
5         if (values[i] == current_node->value) {
6             result[i] = true;
7             break;
8         }
9     }
10    current_node = current_node->next;
11 }
```

This code might seem less optimal than the original code, but it has a much higher ILP and it is typically much faster. The innermost loop over i can be vectorized as well, which gives it an additional speed boost. We explore this in more detail in the experiments section.

42.4.1.2 Binary Trees

When looking up N values in a binary tree, instead of performing N individual lookups, we perform lookups for all N values at once. For this we need a temporary array to store the current node for each of the N values. You will need a counter called `not_null_count` that stores how many values the search hasn't finished. When the `not_null_count` becomes 0, the search has finished for all N nodes. Here is the code snippet:

```

1 // Initialize the current_nodes to root for
2 // each of the N values
3 for (int i = 0; i < N; i++) {
4     current_nodes[i] = root;
5 }
```

```

6
7 do {
8     not_null_count = 0;
9     for (int i = 0; i < N; i++) {
10        if (current_nodes[i] != nullptr) {
11            NodeType* node = current_nodes[i];
12            if (values[i] < node->value) {
13                current_nodes[i] = node->left;
14            } else if (values[i] > node->value) {
15                current_nodes[i] = node->right;
16            } else {
17                current_nodes[i] = nullptr;
18                result[i] = true;
19            }
20            not_null_count++;
21        }
22    }
23 } while (not_null_count > 0);

```

In addition to an increase in instruction-level parallelism, this approach also has a better memory locality. The first few iterations of the `do while` loops will completely hit the L1 cache, the following few iterations will completely hit the L2 cache, etc. Only the last few iterations will not hit any cache.

For this approach, however, the tree must be balanced. The number of iterations of the `do while` loop is equal to the tree depth, therefore, with an unbalanced tree, the additional overhead of the inner `for` loop will probably not pay off.

In the experiment section we measure the effect of interleaving additional work on a binary tree.

42.4.2 Breaking Pointer Chains

Another technique to increase ILP is breaking pointer chains. If we somehow manage to break those chains, even partially or conditionally, this can result in a speedup.

42.4.2.1 Linked Lists

Take for example that we have to lookup N values in a linked list. With N values, we would need to traverse the list N times. But because of the dependency chains, the traversal code will be slow. To break the dependency chain, we traverse the linked list once and create an index: an array of node pointers where a pointer at location X points to the node at location X in the linked list.

In the next step, we don't iterate the linked list, instead, we iterate the index array. Here is the code snippet that traverses the list:

```

1 node* current_node = head;
2
3 // Create an index array
4 while (current_node != nullptr) {

```

```

5     index_vector.push_back(current_node);
6     current_node = current_node->next;
7 }
8
9 // Iterate the index array
10 for (int i = 0; i < values.size(); i++) {
11     for (int j = 0; j < index_vector.size(); j++) {
12         if (index_vector[j]->value == values[i]) {
13             result[i] = true;
14             break;
15         }
16     }
17 }
```

42.4.2.2 Open Addressing Hash Maps

In software development, two common approaches to collision resolution are separate chaining and open addressing. With separate chaining, when a collision occurs, a linked list of all values belonging to the same bucket is formed. With open addressing, when a collision occurs, the runtime searches the next available buckets in the backing array to store the new value.

When the runtime accesses a collisioned bucket, the performance of open addressing hash map will be faster. There are two reasons for this. The first one is better data locality for the open addressing hash map: there is a high chance that two values corresponding to the same bucket are also in the same cache line. And the second is increased ILP: with open-addressing hash maps, the CPU can issue several parallel loads at once. The reason is same as with linked lists from previous section: the collisioned values form a linked list, and iterating linked list is following a chain of dependencies.

42.4.2.3 Binary Trees

For binary trees, breaking pointer chains can be done in several ways. One way is to store a binary tree in an array. In this case, the address of the left and right nodes can be calculated by applying a mathematical formula: if the index of the current node in the array is i , then the index of the left node is $2 * i + 1$ and the address of the right node is $2 * i + 2$. In the experiment section we talk about this.

Binary search in the sorted array is typically faster than a lookup in a binary tree, although the complexity of both lookups is the same $O(\log n)$. There are two reasons for this: smaller dataset size (binary trees store only values, no pointers) and increased ILP.

42.4.3 Shortening the Dependency Chain

Making the dependency chain shorter will result in speedup. In the case of a linked list, we can use unrolled linked lists where we store more than one value inside a single linked list node. In the case of trees, replacing binary trees with N-ary trees (e.g. B* trees) shortens the dependency chain (because the tree is shallower).

Both these techniques increase the available ILP: there is more work to be done when processing a single node, and the processing of a single node mostly has good ILP. Also, these techniques increase the

locality of reference, which means more data cache hits, less memory traffic, and faster speeds.

42.5 Experiments

In this section we will experiment with the various algorithms and ILP. Instruction level parallelism is not possible to measure directly, and also, changing the data layout would result in a different data cache hit rate which would ruin the ILP measurements. Therefore we will take a different approach: we keep the memory layout identical and only change the algorithm.

We measure the runtime, instruction count, and cycles-per-instruction metric (CPI) to get an idea of what is going on. We measure runtime because this is at the end what interests us. We measure instruction count because sometimes the difference in speed can be attributed to a different instruction count and not an improvement in ILP. And finally, CPI metric tells us about hardware efficiency: the smaller the number, the more efficient the algorithm.

All codes are available in our repository⁴.

42.5.1 Interleaving Lookups in a Binary Tree

For the first test of the effect of instruction-level parallelism we use the interleaved binary tree lookup we presented earlier. We use the simple algorithm that works on one-by-one value and we contrast it with the interleaved version. We measure runtimes, instruction count and CPI metric for various binary tree sizes. In both algorithms, the tree's memory layout remains the same, only the algorithm changes.

We ran the tests on various binary tree sizes. The sizes of binary trees in nodes are: 8K, 16K, 32K, ..., 8M, 16M. For each binary tree size we perform 16M lookups in total. This is to get us the feeling about how the runtime is connected to the dataset size.

The first important thing to notice is the instruction count. Interleaved version executes much more instruction than the simple version. For the smallest tree size, it executes 1.8 times more instructions. For the largest tree size, it executes 1.95 times more instructions.

Here are the runtimes of both algorithms depending on the binary tree sizes: (fig. 42.2)

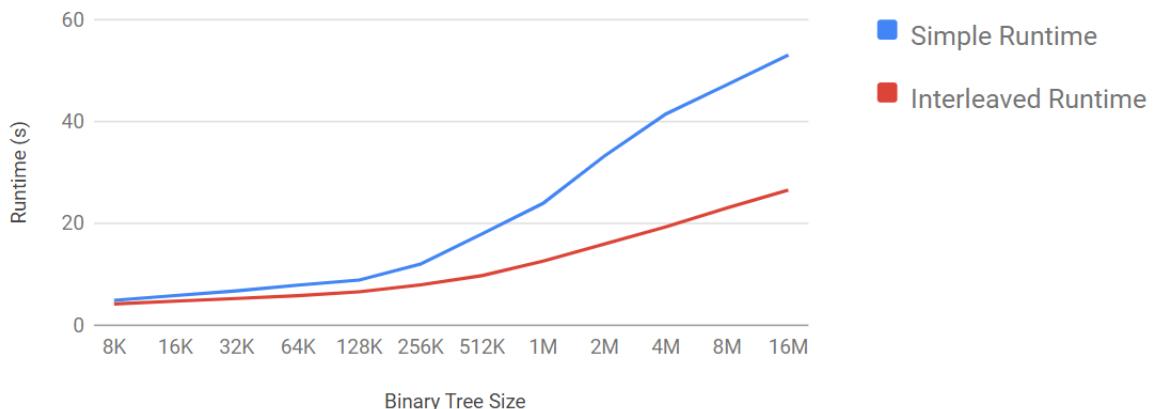


图 42.2: Time needed to perform 16M lookups into a binary tree

⁴<https://github.com/ibogosavljevic/johnysswlab/tree/master/2022-06-instructionlevelparallelism>

For small tree sizes, there isn't much of a difference, but the interleaved version is always faster. But as the tree grows, so the relative difference in speed does. When the tree is the largest, the interleaved version is about 2 times faster than the simple version, even though it is executing almost two times as many instructions.

Let's measure the hardware efficiency. We measure CPI (cycles-per-instruction): the smaller CPI, the better. Here is the graph: (fig. 42.3)

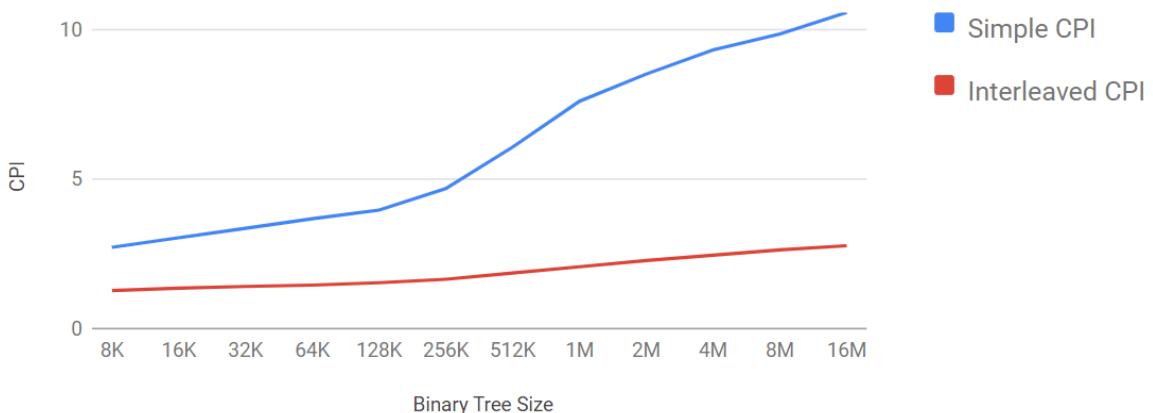


图 42.3: Cycles-per instruction (CPI) for lookups into a binary tree

Two things to observe: interleaved version has a much better CPI and CPI grows as the binary tree grows. CPI for the simple version grows very rapidly: from the initial value of 2.72 it grows to 10.6. The growth of the interleaved version is much smaller: from the initial value of 1.27 it grows to 2.77.

I also wanted to plot the peak memory data load rates. The memory subsystem has a peak memory throughput. If our program reaches it, that means that the memory subsystem is the bottleneck and we need to figure out ways how to use it more efficiently. But if the peak is not reached and yet we see that the program is memory-bound, the problem can be because of the low ILP.

Unfortunately, I couldn't record the data because the hardware counters kept overloading and the data made no sense. What I expected to see was that the interleaved version saturates the memory bus, and the simple version doesn't. But I couldn't confirm this.

42.5.2 Breaking Dependencies with Array-Based Binary Tree

For the second test, we store the binary tree in an array. When stored like this we can use arithmetics to calculate the place of the left or the right child. If the node is at position x in the array, its left node is at position $2 * x + 1$ and its right node is at position $2 * x + 2$. But, in addition to storing the values, we also store left and right pointers. So, there are two ways to perform lookups in the tree: the first one is using arithmetics, as we just explained, and the second one is following the pointers. The data structure remains the same.

With regards to the number of executed instructions, the array-based version executes 1.42 times more instructions than the pointer-based version for the smallest tree and 1.40 times more instructions for the largest tree.

Here is the graph depicting runtimes for pointer-based and array-based lookup in the identical binary tree: (fig. 42.4)

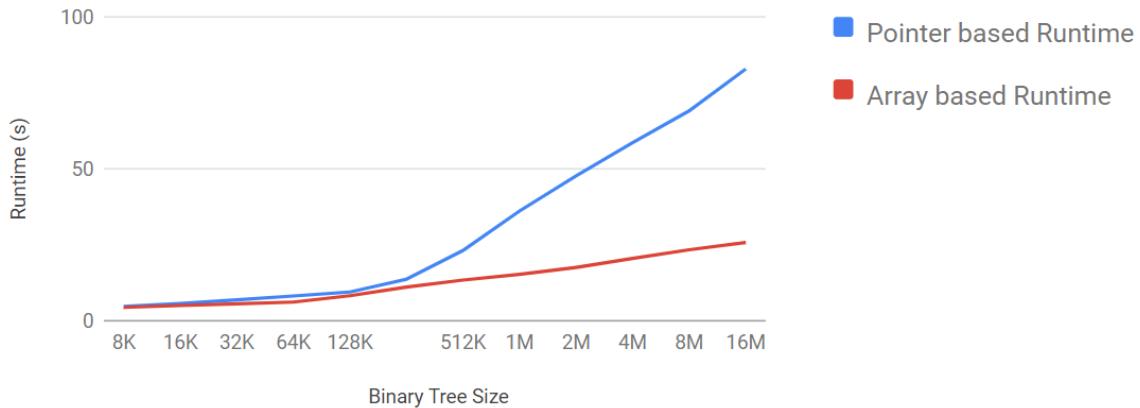


图 42.4: Time needed to perform 16M lookups into a binary tree

The runtimes are more or less the same until the tree has 256K nodes. From that point, the pointer-based array becomes much slower. Since a single node has 24 bytes, the total size of the data structure is 6MB. The total size of LLC on the chip we ran test is 6MB, so the performance problems really hit in once the tree doesn't fit the last level cache anymore.

Let's look at the hardware efficiency through CPI metric: (fig. 42.5)

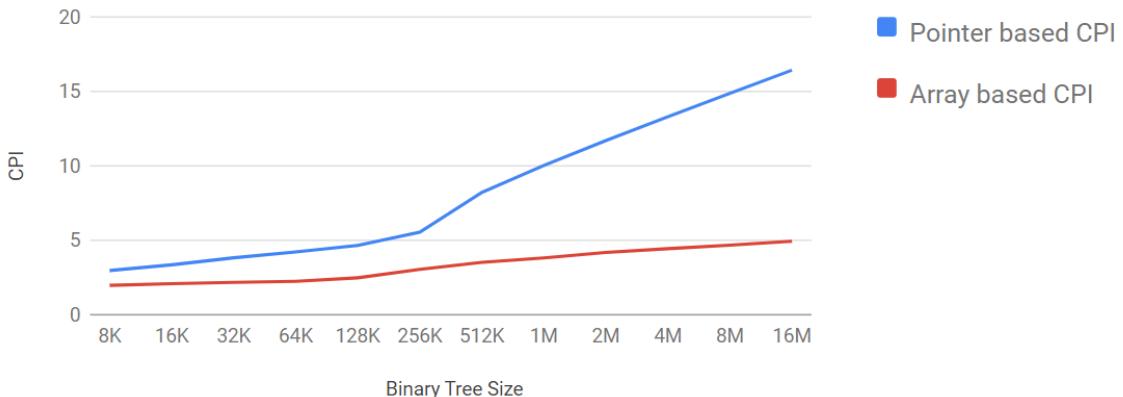


图 42.5: Cycles-per instruction (CPI) for lookups into a binary tree

Similar to the previous experiment, once the dataset doesn't fit the LL cache, the CPI gets much worse for the pointer-based binary tree. When the tree is the largest, the efficacy of the pointer-based version is 3.3 times lower than the array-based version.

42.5.3 Breaking Dependencies in a Linked List

For this experiment we use a hybrid linked list: it is a linked list with a maximum capacity, backed by an array. We also introduce a perfect memory ordering: all the nodes that are neighbors in the linked list are also neighbors in memory. Or in other words: `addr(next_node) = addr(current_node + 1)`. We talked about perfect memory ordering our quest for the fastest linked list⁵.

⁵<https://johnnysswlab.com/the-quest-for-the-fastest-linked-list/>

We use several lookup algorithms to lookup values stored in a vector in the linked list. They are:

- Simple: the simplest algorithm takes a value, and iterates the linked list until it finds it or reaches the end of the list. If there are N values to look up, it will iterate the linked list N times.
- Interleaved: interleaved version performs N searches simultaneously. It takes the first node of the list, and performs N comparisons for the first node, then N comparisons for the second node, etc.
- Array: doesn't iterate the linked list, instead it iterates the underlying array that backs it up. Deleted nodes in the list will have value -1 for the next pointer, so those nodes are skipped.
- Index: it first creates an index array of the linked list as explained in the section Breaking pointer chains. Then it iterates through the index array instead of the linked list.

So, interleaved version would correspond to interleaving additional work and array and index version would correspond to breaking the dependency chain.

The list used for testing has 16M nodes, and we are looking up a total of 128 values. Half of the values do not exist in the linked list, so the lookup algorithm would need to traverse the whole list.

As in the previous examples, we measured runtime, instruction count and CPI for all four types of linked list. We used two linked list memory layouts: the perfect memory layout and a layout where about 25% of the nodes are randomly misplaced. Here are the results:

The performance of various linked list lookup algorithms				
Memory layout	Simple	Interleaved	Array	Index
Perfect layout	Runtime: 2.74 s Instr: 7817 M CPI: 1.19	Runtime: 0.69 s Instr: 10855 M CPI: 0.25	Runtime: 1.79 s Instr: 12507 M CPI: 0.47	Runtime: 2.16 s Instr: 9515 M CPI: 0.71
Misplaced 25% of the nodes	Runtime: 53.9 s Instr: 7756 M CPI: 13.16	Runtime: 1.26 s Instr: 10855 M CPI: 0.38	Runtime: 1.79 s Instr: 12507 CPI: 0.47	Runtime: 17.11 s Instr: 9442 M CPI: 1.89

This table contains a lot of interesting data. Let's investigate it:

- Perfect layout vs misplaced 25% layout: the change in the data layout has an extreme effect on performance for simple and index versions. The simple version is 19.7 times slower and the index version is 7.9 times slower. This can be completely attributed to data cache misses.
- Simple vs index version: the difference in speed is mostly due to the increase in instruction-level parallelism. Even though the index version executes more instructions, for perfect layout, the index version is 1.27 times faster. For the misplaced layout, the version is 3.15 times faster.
- Array version doesn't care for memory layout: the reason is that the array version scans the underlying array from left to right in both cases for both versions, so nothing really changes.
- Interleaved version is the fastest: the reason is quite simple. The innermost loop in the interleaved version is a simple for loop iterating over a vector of integers. From the hardware efficiency point of view, this is very efficient. Those loops also can be vectorized⁶ for an additional speed boost, although this didn't happen in our case.

⁶<https://johnnysswlab.com/crash-course-introduction-to-parallelism-simd-parallelism/>

42.6 Conclusion

This post has been very exciting for me to write, because it throws a different light on hardware and how instruction-level parallelism can help us write faster programs.

The crucial note here is memory load dependency: in the presence of high data cache miss rates, memory load dependency can really cripple the software performance. Knowing when load dependencies happen and how to break the dependency chain can help you write much faster software.

With regards to which technique to use, to me, it seems that interleaving additional work is both simpler and more effective than breaking dependency chains. Breaking dependency chains requires a rewrite of both the data structure and the algorithm. Breaking dependency chains is also more difficult to achieve, because those techniques bring come with limitations that make them difficult to use (e.g. inserting a value in an array represented binary tree is not simple, also, not every linked list can be represented as an array).

When there is a load dependency chain but a good memory layout (e.g. perfect memory layout with linked lists), then the benefits of increased ILP are limited.

Part VI

Performance

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

——Donald Knuth

本 Part 主要包含与 C++ 性能相关的多个方面，例如，编译期编程、内存管理、编译器调优、代码重构……

作为一门强调性能的语言，C++ 含有许多跟性能相关的特性，尤其是 Modern C++ 的诸多新特性。作为一种贴近硬件而又提供大量零开销抽象机制的编程语言，C++ 也为开发人员提供了大量的性能优化潜力。优化的基本原则对于所有编程语言来说都差不多，这其中最基本的一条是：

不要执行不必要的代码！

一般而言，对于会损伤代码可读性、可扩展性和可移植性的优化应该放到最后，并且最好把它们封装起来，以防影响程序的其他部分。

进入 Modern C++ 的新纪元，尤其是当 *constexpr*、*concepts*、*constinit* 等一干新特性地提出，可以看出性能优化（这里特指的是程序运行期）已经从传统的一些做法（比如压榨 CPU、操纵内存、控制圈复杂度等），逐渐升级转向为将计算放入程序编译期（元编程），进而提升运行期性能的趋势。

性能是一项系统工程，性能调优之路也没有终点。但无论如何，请记住：“把正确的代码改快速”要比“把快速的代码改正确”容易太多。性能的提升不是立足于感觉和猜测，必须基于优化前后的真实数据说话。

通过本 Part，读者可以了解到 C++ 开发中提升性能的一些技术，深入理解高阶 C++ 程序设计的思路和方法，梳理自己的知识体系，并实际应用这些知识来改进和提升程序性能。

2023 年 4 月 30 日
何荣华

Chapter 43

5 Ways to Initialize a String Member

• 里缪 曲 2022-11-20 四★★★

C++ 初始化成员的方式有许多，尤其是随着 C++11 值类别的重新定义，各种方式之间的差异更是细微。本文将以 String 成员初始化为例，探讨以下 5 种方式之间的优劣：

- call by-const-reference
- call by-value
- two-overloads
- std::string_view
- forwarding references

输入不同，它们的开销也完全不同，我们将以 4 种不同的输入分别讨论。本篇结束，下方的表格也将填满。

	call by-const-reference	call by-value	Two-overloads	string_view	Forwarding reference
Implicit ctor					
lvalue					
xvalue					
rvalue					

实际要讨论的情况远超 1 种，因此这张表格在不同的情境下，填入的开销也不尽相同。多种情况，多个概念，交叉讨论，错综复杂，这也是本篇文章难度能入四星的原因。

下面正式进入讨论。

43.1 call by-const-reference

这种方式最是广为流传，C++11 之前亦可使用，是早期的推荐方式。比如：

```
struct S {  
    std::string mem;
```

```
S(const std::string& s) : mem{s} {}  
};
```

即便现在，使用这种方式也是大有人在。根据 4 种不同的输入分析其开销，代码如下：

```
std::string str {"dummy"};  
S s1("dummy");           // 1. Implicit ctor  
S s2(str);              // 2. lvalue  
S s3(std::move(str));    // 3. xvalue  
S s4(std::string{"dummy"}); // 4. prvalue
```

第一，Implicit ctor。当传入一个字符串字面量时，会先通过隐式构造创建一个临时的 string 对象，将它绑定到形参之上，再通过拷贝构造复制到成员变量。共 2 次分配。

第二，lvalue。对于左值，将直接绑定到形参上，再通过拷贝构造复制到成员变量。共 1 次分配。

第三，xvalue。对于消亡值，操作同上。共 1 次分配。

第四，prvalue。对于纯右值，操作同上。共 1 次分配。

由此可知，使用 const-reference string 时，至少存在 1 次分配。对于左值来说，这本无可厚非；但对于右值来说，这将徒增一次没必要的拷贝；对于字符串字面量，还会由隐式构造创建一个临时对象，增加一次开销。

因此，这种方式只有针对左值时，效果才不错，其他情况都会增加一次开销。

43.2 call by-value

这是从 C++11 开始也广为流传的一种方式，使用的人也不少。4 种调用不变，实现变为了：

```
struct S {  
    std::string mem;  
    S(std::string s) : mem{std::move(s)} {}  
};
```

这种方式采用值传递参数，在许多人的印象中，这种开销很大，实际情况到底如何呢？

第一，Implicit ctor。同样，先通过隐式构造创建一个临时对象，然后将其值偷取到成员变量。共 1 次分配 +1 次移动。

第二，lvalue。拷贝对象，然后将其值偷取到成员变量。共 1 次分配 +1 次移动。

第三，xvalue。值经过两次偷取到成员变量。共 0 次分配 +2 次移动。

第四，prvalue。值直接原地构造，然后偷取到成员变量。共 0 次分配 +1 次移动。

可以看到，call by-const-reference 的缺点，这种方式全部避免。只在接受左值时，多了一次移动。

因此，很多情况下，这种方式往往是一种更好的选择，它可以避免无效的拷贝。

43.3 Two-overloads

这种方式通过多提供一个移动构造来消除 call by-const-reference 的缺点，由于存在两个重载函数，所以称为 two-overloads。此时实现变为了：

```
struct S {
    std::string mem;
    S(const std::string& s) : mem{s} {}
    S(std::string&& s) : mem{std::move(s)} {}
};
```

相信你已经猜到了现在的开销。

第一, Implicit ctor。同样, 先创建一个临时对象, 然后调用移动构造。共 1 次分配 +1 次移动。

第二, lvalue。调用拷贝构造。共 1 次分配。

第三, xvalue。调用移动构造。共 0 次分配 +2 次移动。

第四, prvalue。调用移动构造。共 0 次分配 +1 次移动。

通过多增加一个重载函数, 得到了不少好处, 因此这也是一种可行的方式, 但多写一个重载函数总是颇显琐碎。

43.4 C++17 string_view

C++17 std::string_view 也是一种可行的方案, 所谓是又轻又快。采用这种方式, 实现变为:

```
struct S {
    std::string mem;
    S(std::string_view s) : mem{s} {}
};
```

此时的开销情况如何?

第一, Implicit ctor。除了 mem 创建, 没有多余开销。共 1 次分配。

第二, lvalue。通过隐式转换创建string_view, 然后拷贝到成员变量。共 1 次分配。

第三, xvalue。同上。共 1 次分配。

第四, prvalue。同上。共 1 次分配。

对于右值, 这种方式也会产生没必要的开销。

最重要的是, std::string_view 隐藏有许多潜在的危险, 就像操作裸指针一样, 需要程序员来确保它的有效性。稍不留神, 就有可能产生悬垂引用, 指向一个已经删除的string 对象。

因此, 若是对其没有一定的研究, 极有可能使用错误的用法。

43.5 Fowarding references

Forwarding references 可以自动匹配左值或是右值版本, 也是一种不错的方式。实现变为:

```
struct S {
    std::string mem;
    template <class T>
    S(T&& s) : mem{ std::forward<T>(s) } {}
};
```

此时的开销又如何?

第一, Implicit ctor。除了 mem 构造, 无额外开销。共 1 次分配。

第二, lvalue。直接绑定到实例化的模板函数参数上, 然后拷贝一份。共 1 次分配。

第三, xvalue。调用移动构造。共 0 次分配 +2 次移动。

第四, prvalue。调用移动构造。共 0 次分配 +1 次移动。

这种方式借助了模板, 参数的实际类型根据 TAD 推导, 所以它的开销也都很小。

很多时候, 这种方式就是最佳选择, 它可以避免非必要的移动或是拷贝, 也适用于非 String 成员的初始化。

但有些时候, 你可能想明确指定参数类型, 此时这种方式就多有不便了。下节有相应例子。

43.6 Long String 成员开销对比

分析至此, 已然可以初步得出一张开销对表。

5 种方式初始化 Long String 成员开销对比表					
	call by-const-reference	call by-value	Two-overloads	string_view	Forwarding reference
Implicit ctor	2 Allocations	1 Allocation + 1 Move	1 Allocation + 1 Move	1 Allocation	1 Allocation
lvalue	1 Allocation	1 Allocation + 1 Move	1 Allocation	1 Allocation	1 Allocation
xvalue	1 Allocation	2 Move	2 Move	1 Allocation	2 Move
prvalue	1 Allocation	1 Move	1 Move	1 Allocation	1 Move

因此, 若要问哪种方式初始化 String 成员比较好, 如何回答? 看情况。没有哪种方式是完全占优的, 可以依据使用次数最多的操作计算消耗, 从而正确决策。举个例子:

```
struct S {
    using value_type = std::vector<std::string>;
    using assoc_type = std::map<std::string, value_type>;

    void push_data(std::string_view key, value_type data) {
        datasets.emplace(std::make_pair(key, std::move(data)));
    }

    assoc_type datasets;
};
```

功能很简单, 就是往一个 map 中添加数据。此时, 如何让浪费最小? 假设我们后面使用次数最多的操作为:

```
1 S s;
2 s.push_data("key1", {"Dear", "Friend"});
3 s.push_data("key2", {"Apple"});
4 s.push_data("key3", {"Jack", "Tom", "Jerry"});
5 s.push_data("key4", {"20", "22", "11", "20"});
```

那么上述实现就是一种较好的方式。

对于键，如果使用 call by-const-reference，将会创建一个没必要的临时对象，而使用`string_view`可以避免此开销。

对于值，实际上也使用隐式构造创建了一个临时`vector`对象，此时 call by-value 也是一种开销较小的方式。

你可能觉得 Forwarding reference 也是一种不错的方式。

```
void push_data(auto&& key, auto&& data) {
    datasets.emplace(std::make_pair(
        std::forward<decltype(key)>(key),
        std::forward<decltype(data)>(data)
    ));
}
```

对于键来说的确不错，但对于值来说就存在问题了。因为模板参数推导为`initializer_list`，而参数传递需要的是`vector`，使用这种方式还得手动创建一个临时的`vector`。所以，具体问题具体分析，才能选择最恰当的方式，有时甚至可以组合使用。

大家也许注意到，开销对比图标题为“初始化 Long String 成员开销图”，那么还有短 String 吗？

43.7 SSO 短字符串优化

各家编译器在实现`std::string`时，基本都会采取一种 SSO(Small String Optimization) 策略。

此时，对于短字符串，将不会在堆上额外分配内存，而是直接存储在栈上。比如，有些实现会在`size`的最低标志位上用 1 代表长字符串，0 代表短字符串，根据这个标志位来决定操作形式。

可以通过重载`operator new`和`operator delete`来捕获堆分配情况，一个例子如下：

```
1 // example fr. https://stackoverflow.com/a/28003328
2 std::size_t allocated = 0;
3
4 void* operator new(size_t sz) {
5     void* p = std::malloc(sz);
6     allocated += sz;
7     return p;
8 }
9
10 void operator delete(void* p) noexcept {
11     std::free(p);
12 }
13
14 int main() {
15     allocated = 0;
16     std::string s("hi");
17     std::printf("stack space = %zu, heap space = %zu, capacity = %zu\n",
18                 sizeof(s), allocated, s.capacity());
19 }
```

在 clang 14.0.0 上得出的结果为：

```
stack space = 32, heap space = 0, capacity = 15
```

可以看到，对于短字符串，将不会在堆上分配额外的内存，内容实际存在在栈上。早期版本的编译器可能没有这种优化，但如今的版本基本都有。也就是说，这时的移动操作实际就相当于复制操作。于是开销就可以如下表。

5 种方式初始化 Short String 成员开销对比表					
	call by-const-reference	call by-value	Two-overloads	string_view	Forwarding reference
Implicit ctor	2 Allocations	2 Allocations	2 Allocations	1 Allocation	1 Allocation
lvalue	1 Allocation	2 Allocations	1 Allocation	1 Allocation	1 Allocation
xvalue	1 Allocation	2 Allocations	2 Allocations	1 Allocation	2 Allocations
prvalue	1 Allocation	1 Allocation	1 Allocation	1 Allocation	1 Allocation

于是可以得出结论：尽管小对象的拷贝操作很快，call by-value 还是要慢于其他方式，`string_view` 则是一种较好的方式。

但是，`string_view` 使用起来要格外当心，若你不想为此操心，使用 call by-const-reference 则是一种不错的方式。

43.8 无拷贝，无移动

当仅需要接受参数，之后即不拷贝，也无移动的情境下，情况又不一样。此时的开销如下表。

5 种方式「仅作为参数情况下」开销对比表					
	call by-const-reference	call by-value	Two-overloads	string_view	Forwarding reference
Implicit ctor	1 Allocation	1 Allocation	1 Allocation	No-cost	No-cost
lvalue	No-cost binding	1 Copy	No-cost binding	No-cost	No-cost binding
xvalue	No-cost binding	1 Move	1 Move	No-cost	1 Move
prvalue	No-cost binding	In-place construct	In-place construct	No-cost	In-place construct

前三种方式对于隐式构造，都会产生一个临时对象，故皆含一次分配。

后两种方式没有这次额外分配。Views 的创建非常之轻，可忽略不计；Forwarding references 经过 TAD 推导的参数为常量串，也无消耗。

总的来说，我们能得出，通常情况下，后两种方式在这种情境下是一种开销更小的方式。若不考虑隐式构造，则除了 call by-value，其他方式是一种开销更小的方式。

43.9 优化限制：Aliasing situations

Aliasing situations 指的是多个变量实际指向的是同一块内存，这些变量之间互为别名。这种情况会导致编译器束手束脚，不敢优化。以引用的方式传递参数便会产生许多额外的 Aliasing situations。举个例子：

```
1 // example fr. https://reducto.dev/cpp/2022/06/27/pass-by-value-vs-pass-by-reference.html
2 int foo_by_ref(const S& s) {
3     int m = s.value;
4     bar();
5     int n = s.value;
6     return m + n;
7 }
8
9 int foo_by_value(S s) {
10    int m = s.value;
11    bar();
12    int n = s.value;
13    return m + n;
14 }
```

引用传递版本，编译器无法判定`bar()`中是否修改了`s.value`，比如`s`引用的是一个全局变量，`bar()`中就可以修改它。因此，`m`和`n`的值可能并不相同，编译器必须加载两次`s.value`。

而值传递版本，由于参数进行了拷贝，不存在外部修改，`m`和`n`的值肯定相同，于是编译器可以优化为只加载一次`s.value`。

这是 call by-const-reference 的另一处缺点，它可能会限制编译器的优化，而这又恰恰成了 call by-value 的一个优点。

43.10 总结

本篇文章介绍了 5 种初始化 String 成员的方式，详细分析对比了它们的开销。

没有哪种方式是最优解，如何选择需要依具体情况而论。

Two-overloads 这种方式一般不会考虑，因为总有其他方式比它的开销更小，还只需编写一个函数。`string_view`在很多情况下的开销可观，但是需要格外注意潜在的悬垂引用问题。其他三种方式亦是有利有弊，可根据文中提及的各种情况进行分析。

总而言之，各方式之间存在着细微而本质的差别，且还有许多特殊情况需要单独分析，开销在不同情境下也不尽相同。

一句话，看情况。

Chapter 44

Using `final` in C++ to improve performance

• Niall Cooling  2022-11-14   

Dynamic polymorphism (virtual functions) is central to Object-Oriented Programming (OOP). Used well, it provides hooks into an existing codebase where new functionality and behaviour can (relatively) easily be integrated into a proven, tested codebase.

Subtype inheritance can bring significant benefits, including easier integration, reduced regression test time and improved maintenance.

However, using virtual functions in C++ brings a runtime performance overhead. This overhead may appear inconsequential for individual calls, but in a non-trivial real-time embedded application, these overheads may build up and impact the system's overall responsiveness.

Refactoring an existing codebase late in the project lifecycle to try and achieve performance goals is never a welcome task. Project deadline pressures mean any rework may introduce potential new bugs to existing well-tested code. And yet we don't want to perform unnecessary premature optimization (as in avoiding virtual functions altogether) as this tends to create technical debt, which may come back to bite us (or some other poor soul) during maintenance.

The `final` specifier was introduced in C++11 to ensure that either a class or a virtual function cannot be further overridden. However, as we shall investigate, this also allows them to perform an optimization known as *devirtualization*, improving runtime performance.

44.1 Interfaces and subtyping

Unlike Java, C++ does not explicitly have the concept of Interfaces built into the language. Interfaces play a central role in Design Patterns and are the principal mechanism to implement the SOLID ‘D’ Dependency Inversion Principle pattern.

44.1.1 Simple Interface Example

Let's take a simplified example; we have a mechanism layer defining a class named `PDO_Protocol`. To decouple the protocol from the underlying utility layer, we introduced an interface called `Data_link`. The

concrete class `CAN_bus` then *realizes* the Interface. (fig. 44.1)

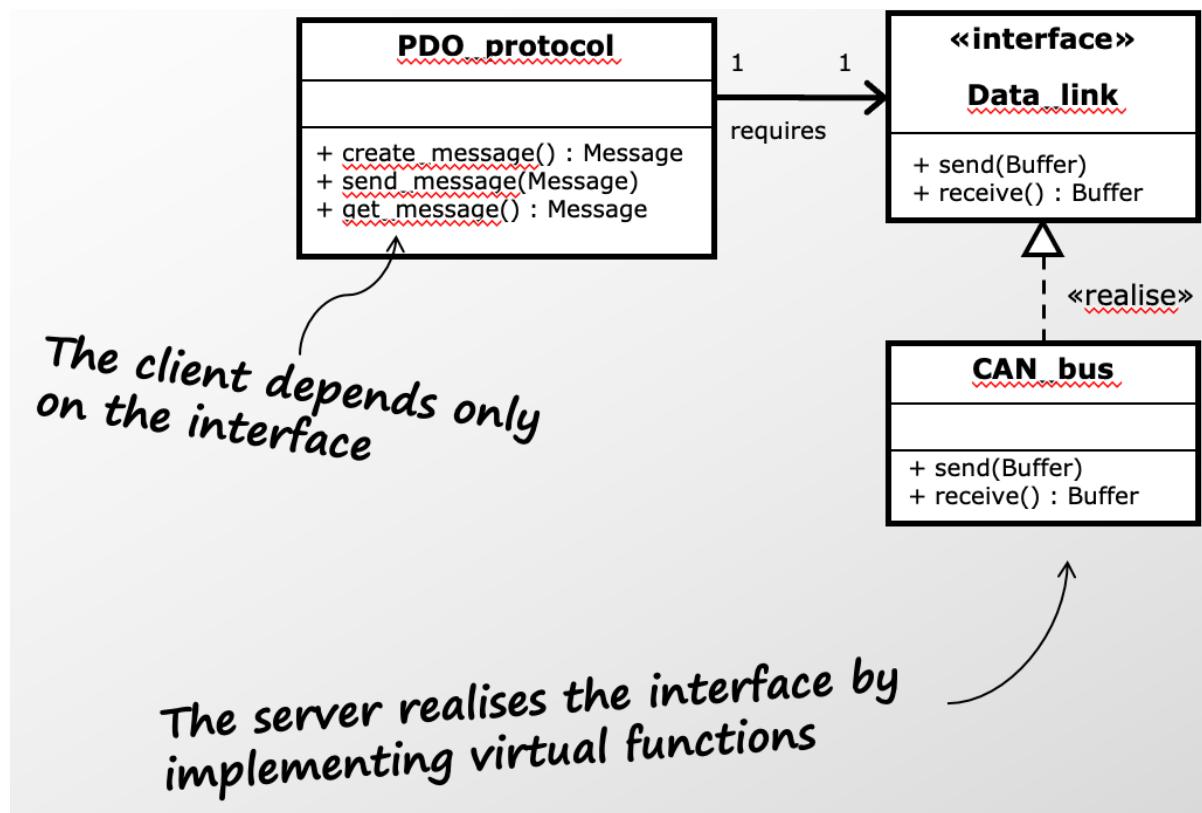


图 44.1: class diagram

This design would yield the following Interface class:

```

1 #pragma once
2 #include "buffer.h"
3 class Data_link {
4 public:
5     virtual void send(Buffer buf) = 0;
6     virtual Buffer receive() = 0;
7     virtual ~Data_link() = default;
8 };
  
```

Side note: I'll park the discussion about using `pragma once`, virtual-default-destructors and pass-by-copy for another day.

The client (in our case, `PDO_protocol`) is only dependent on the Interface, e.g.

```

1 #pragma once
2 #include "message.h"
3
4 class Data_link;
5
6 class PDO_protocol {
7 public:
  
```

```

8     PDO_protocol(Data_link &transport);
9
10    Message create_message();
11    void send_message(Message msg);
12    Message get_message();
13
14 private:
15     Data_link *link;
16 };
17
18 void PDO_protocol::send_message(Message msg) {
19     // ...
20     auto buf = msg.get_buffer();
21     // ...
22     link->send(std::move(buf));
23     // ...
24 }
25
26 Message PDO_protocol::get_message() {
27     // ...
28     auto buf = link->receive();
29     // ...
30     return Message{std::move(buf)};
31 }
```

Any class *realizing* the Interface, such as `CAN_bus`, must `override` the pure-virtual functions in the Interface:

```

1 #pragma once
2 #include "data_link.h"
3
4 class CAN_bus : public Data_link {
5 public:
6     CAN_bus() = default;
7     ~CAN bus() override = default;
8 protected:
9     void send(Buffer buf) override;
10    Buffer receive() override;
11 private:
12     // implememation stuff
13 };
```

Finally, in `main`, we can *bind* a `CAN_bus` object to a `PDO_protocol` object. The calls from `PDO_protocol` invoke the overridden functions in `CAN_bus`.

```

1 #include "CAN_bus.h"
2 #include "PDO_protocol.h"
3
4 int main() {
5     CAN_bus can_bus{};
6     PDO_protocol protocol{can_bus};
7
8     auto msg = protocol.create_message();
9     protocol.send_message(std::move(msg));
10 }
```

44.1.1.1 Using dynamic polymorphism

It then becomes very straightforward to swap out the `CAN_bus` for an alternative `utility` object, e.g. `RS422`:

```

1 #pragma once
2 #include "data_link.h"
3
4 class RS422 : public Data_link {
5 public:
6     RS422() = default;
7     ~RS422() override = default;
8 protected:
9     void send(Buffer buf) override;
10    Buffer receive() override;
11
12 private:
13     // implememation stuff
```

In `main`, we bind the `PDO_protocol` object to the alternative class.

```

1 #include "RS422.h"
2 #include "PDO_protocol.h"
3
4 int main() {
5     RS422 serial_bus{};
6     PDO_protocol protocol{serial_bus};
7
8     auto msg = protocol.create_message();
9     protocol.send_message(std::move(msg));
10 }
```

Importantly, there are no changes to the `PDO_protocol` class. With appropriate unit testing, introducing the `RS422` code into the existing codebase involves integration testing (rather than a blurred unit/integration test).

There are many ways we could create the new type (i.e. using factories, etc.), but, again, let's park that for this post.

44.2 The cost of Dynamic Polymorphic behaviour

Using subtyping and polymorphic behaviour is an important tool when trying to manage change. But, like all things in life, it comes at a cost.

The code generated in the examples using the Arm GNU Toolchain v11.2.1.

A previous posting covered the Arm calling convention for AArch32 ISA. For a simple member-function call, e.g.:

```

1  class Sensor {
2  public:
3      double get_value();
4      void set_ID(int);
5  private:
6      // Sensor Data
7  };
8
9  void read_sensor(Sensor& sensor) {
10     sensor.get_value();
11 }
12
13 int main() {
14     sensor sensor_1{};
15     sensor sensor_2{};
16     read_sensor(sensor_1);
17 }
```

We get the following assembler for the call to the member function in `read_sensor`:

```
bl    Sensor::get_value()
```

The `Branch with Link (bl)` opcode is the AArch32 function calling convention (`r0` contains the object's address).

So what happens at the call site when we make this function virtual?

```

1  class Sensor {
2  public:
3      virtual double get_value();
4      virtual void    set_ID(int);
5  private:
6      // Sensor Data
7  };
8
```

```

9 void read_sensor(Sensor& sensor) {
10     sensor.get_value();
11 }

```

The generated assembler for `sensor.get_value()` becomes:

```

ldr    r3, [r0]
ldr    r3, [r3]
mov    lr, pc
bx    r3

```

The actual code generated, naturally, depends on the specific ABI (Application Binary Interface). But, for all C++ compilers, it will involve a similar set of steps. Visualizing the implementation: (fig. 44.2)

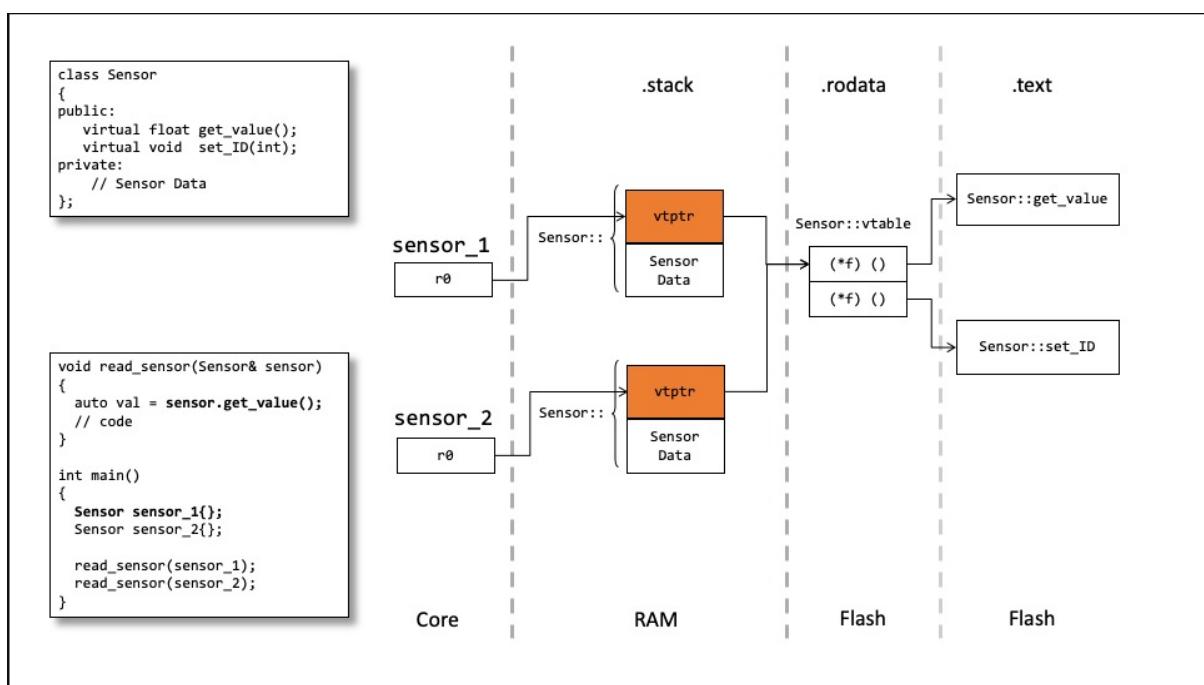


图 44.2: vtable image

And examining the generated assembler, we can deduce the following behaviour:

- `r0` contains the address of the object (passed as the parameter to `read_sensor`)
- the contents at this address are loaded into `r3`
- `r3` now contains the vtable-pointer (`vtptr`)
- The `vtptr` is, in effect, an array of function pointers.
- The first entry in the vtable is loaded back into `r3` (e.g. `vtable[0]`)
- `r3` now contains the address of `Sensor::get_value`
- the current program counter (`pc`) is moved into the link register (`lr`) before the function call
- The `branch-with-exchange` opcode is executed. So, the instruction `bx r3` calls `Sensor::get_value`

If, for example, we were calling `sensor.set_ID()`, then the second memory load would be `LDR r3,[r3,#4]` to load the address of `Sensor::set_ID` into `r3` (e.g. `vtable[1]`). Most ABIs structure the `vtable` based on the order of virtual function declaration.

We can deduce that the overhead of using a virtual function (for Arm Cortexv7-M) is:

- 2 x LDR
- 1 x MOV

However, what is significant is the second memory load (`LDR r3,[r3]`), as this memory read requires Flash access. A read from Flash is typically slower than an equivalent read from SRAM. A lot of design effort goes into improving Flash read performance, so your “mileage may vary” regarding the actual timing overhead.

44.2.1 Using polymorphic functions

If we create a class that derives from `Sensor`, e.g.

```

1  class Rotary_encoder : public Sensor {
2  public:
3      float get_value() const override;
4
5  private:
6      // Encoder Data;
7 }
```

And then pass an object of the derived type to the function `read_sensor`, then the same assembler is executed.

```

1  int main() {
2      Sensor sensor{};
3      read_sensor(sensor);
4
5      Rotary_encoder encoder{};
6      read_sensor(encoder);
7 }
```

But by visualizing the memory model, it becomes clear how the same code:

```

ldr    r3, [r0]
ldr    r3, [r3]
mov    lr, pc
bx    r3
```

Invokes the derived function: (fig. 44.3)

The derived class has its own `vtable` populated at link-time. Any overridden functions replace the `vtable` entry with the address of the new function. The constructors are responsible for storing the address of the `vtable` in the classes `vtpr`.

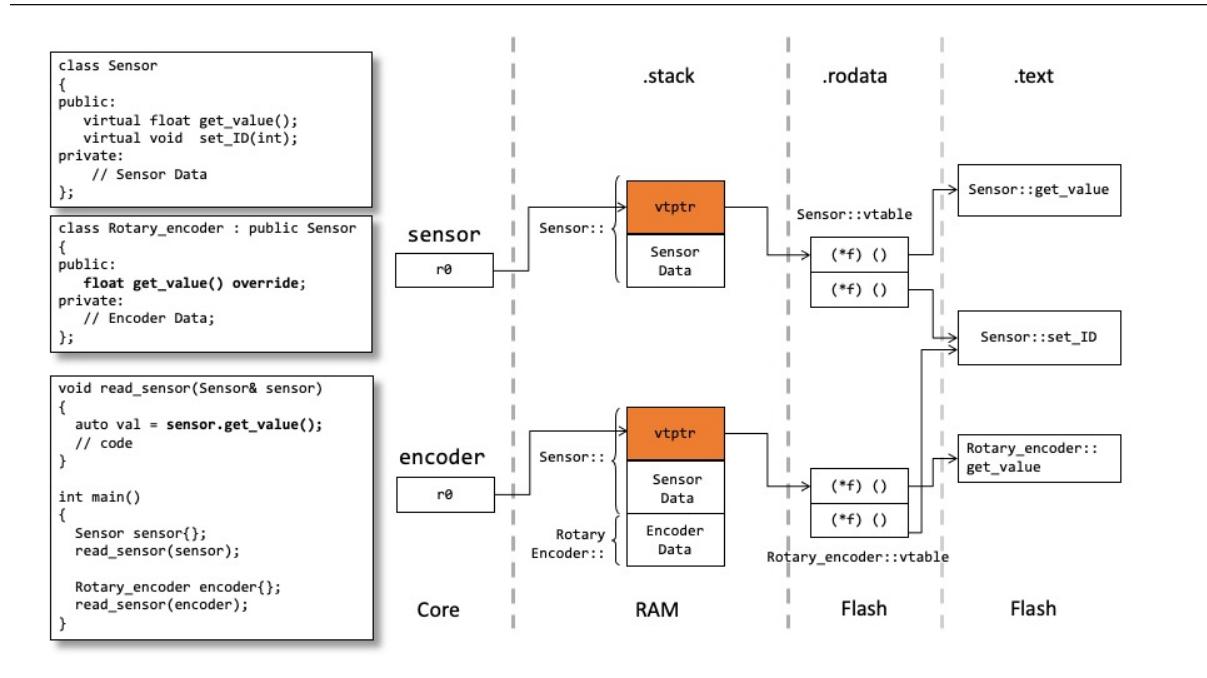


图 44.3: derived vtable image

Any virtual functions in the base class that are not overridden still point at the base class implementation. Pure-virtual functions (as used in the interface pattern) have no entry populated in the `vtable`, so they must be overridden.

44.3 Introducing final

As previously noted, the `final` specifier was introduced alongside `override` in C++11.

The `final` specifier was introduced to ensure that a derived class cannot override a virtual function or that a class cannot be further derived from it.

For example, currently, we could derive further from the `Rotary_encoder` class.

```

1  class CAS60_encoder : public Rotary_encoder
2  {
3  public:
4      float get_value() override;
5  private:
6      // Encoder Data;
7 };

```

When defining the `Rotary_encoder` class, this may not have been our intended design. Adding the `final` specifier stops any further derivation.

A class, may be specified as `final` e.g.

```

1  class Rotary_encoder final : public Sensor
2  {
3  public:

```

```

4     float get_value() override;
5 private:
6     // Encoder Data;
7 };

```

Which, if inherited from, generates the following error: (fig. 44.4)

```

<source>:18:7: error: cannot derive from 'final' base 'Rotary_encoder' in
derived type 'CAS60_encoder'
  18 | class CAS60_encoder : public Rotary_encoder
      | ^~~~~~
Compiler returned: 1

```

图 44.4: final class error

Or an individual function can be tagged as **final**, e.g.:

```

1  class Rotary_encoder : public Sensor
2  {
3      public:
4          float get_value() override;
5      private:
6          // Encoder Data;
7  };
8
9  class CAS60_encoder : public Rotary_encoder
10 {
11     public:
12         float get_value() override;
13     private:
14         // Encoder Data;
15 };

```

```

<source>:21:10: error: virtual function 'virtual float
CAS60_encoder::get_value()' overriding final function
  21 |     float get_value() override;
      | ^~~~~~
<source>:13:10: note: overridden function is 'virtual float
Rotary_encoder::get_value()'
  13 |     float get_value() override final;
      | ^~~~~~
Compiler returned: 1

```

图 44.5: final func error

44.4 Devirtualization

Okay, so how can this help with compiler optimization?

When calling a function, such as `read_sensor` and the parameter is a pointer/reference to the Base class, which in turn calls a virtual member function, the call *must* be polymorphic.

If we overload `read_sensor` to take a `Rotary_encode` object by reference, e.g.

```

1 void read_sensor(Sensor& sensor)
2 {
3     auto val = sensor.get_value();
4     // code
5 }
6 void read_sensor(Rotary_encoder& sensor)
7 {
8     auto val = sensor.get_value();
9     // code
10}
11
12 int main()
13 {
14     Sensor sensor{};
15     read_sensor(sensor); // calls read_sensor(Sensor&)
16
17     Rotary_encoder encoder{};
18     read_sensor(encoder); // calls read_sensor(Rotary_encoder&)
19 }
```

If the compiler can prove exactly which *actual* method is called at compile time, it can change a virtual method call into a direct method call.

Without the `final` specifier, the compiler cannot *prove* that the `Rotary_encode` reference, `sensor`, isn't bound to a further derived class instance. So the generated assembler for both `read_sensor` functions are identical.

However, if we apply the `final` specifier to the `Rotary_encoder` class, the compiler can *prove* that the only matching call must be `Rotary_encoder::get_value`, then it can apply `devirtualization` and generate the following code for `read_sensor(Rotary_encoder&)`:

```
b1   Rotary_encoder::get_value()
```

44.4.1 Templates and final

As our two `read_sensor` functions are identical, the DRY principle comes into play. If we modify the code so that `read_sensor` is a template function, e.g.

```

1 template <typename T>
2 void read_sensor(T&& sensor)
```

```

3  {
4      auto val = sensor.get_value();
5      // code
6  }

```

The code generator will bind dynamically or statically as appropriate, depending on whether we call with a `Sensor` object or a `Rotary_encoder` object.

44.4.2 Revising the Interface

Given the potential for `devirtualization`, can we utilize this in our Interface design?

Unfortunately, for the compiler to be able to *prove* the actual method call, we must use `final` in conjunction with a pointer/reference to the derived type. Given the original code:

```

1 #pragma once
2 #include "message.h"
3
4 class Data_link;
5
6 class PDO_protocol {
7 public:
8     PDO_protocol(Data_link &transport);
9
10    Message create_message();
11    void send_message(Message msg);
12    Message get_message();
13
14 private:
15    Data_link *link;
16 };

```

The compile cannot perform `devirtualization` because it has a reference to the interface (base) class and not the derived class. This leaves us with two potential refactoring solutions:

- Modify the link type to the derived type
- Make the client a template class

44.4.2.1 Devirtualization using a direct link

Using a direct link is a “quick and dirty” fix.

```

1 #pragma once
2 #include "message.h"
3
4 // class Data_link;
5 class CAN_bus;

```

```

6
7 class PDO_protocol {
8 public:
9     PDO_protocol(CAN_bus &transport);
10
11    Message create_message();
12    void send_message(Message msg);
13    Message get_message();
14
15 private:
16    CAN_bus *link;
17 };

```

It does change the `PDO_protocol` header, but otherwise, it “does the job”. The generated code now calls `CAN_bus::send` and `CAN_bus::recieve` directly rather than through a `vtable` call.

However, using this approach, we reintroduce the coupling between the “Mechanism layer” and the “Utility layer”, breaking the DIP.

44.4.2.2 Devirtualization using templates

Alternatively, we can rework the client code as a template class, where the template parameter specifies the link class.

```

1 #pragma once
2 #include "message.h"
3
4 template <typename Data_link>
5 class PDO_protocol {
6 public:
7     PDO_protocol(Data_link &transport);
8
9     Message create_message();
10    void send_message(Message msg);
11    Message get_message();
12
13 private:
14    Data_link *link;
15 };
16
17 template <typename Data_link>
18 PDO_protocol<Data_link>::PDO_protocol(Data_link &transport)
19     : link{&transport} {}
20
21 template <typename Data_link>
22 Message PDO_protocol<Data_link>::create_message() {

```

```

23     return Message{};
24 }
25
26 template <typename Data_link>
27 void PDO_protocol<Data_link>::send_message(Message msg) {
28     // ...
29     link->send(std::move(msg.get_buffer()));
30     // ...
31 }
32
33 template <typename Data_link>
34 Message PDO_protocol<Data_link>::get_message() {
35     // ...
36     auto buf = link->receive();
37     // ...
38     return Message{std::move(buf)};
39 }
```

Templates bring their complications, but it does ensure we get static binding to any classes specified as `final`.

44.5 Summary

The `final` specifier offers an opportunity to refactor existing interface code to alter the binding from dynamic to static polymorphism, typically improving runtime performance. The actual gains will depend significantly on the underlying ABI and machine architecture (start throwing in pipelining and caching, and the waters get even muddier).

Ideally, when using virtual functions in embedded applications, considering whether a class should be specified as `final` should be decided at design time rather than late in the project's timeline.

Chapter 45

Technique: Compile Time Code Generation and Optimization

👤 Jonathan Müller 📅 2022-1-27 💬 ★★★

C++ `constexpr` is really powerful. In this blog post, we'll write a compiler that can parse a Brainfuck program given as string literal, and generate optimized assembly instructions that can then be executed at runtime. The best part: we neither have to actually generate assembly nor optimize anything ourselves! Instead we trick the compiler into doing all the hard work for us.

The same technique can be used whenever you want to specify some sort of “program” in a different way and translate it at runtime: regexes, routing tables, etc.

45.1 Brainfuck

Brainfuck is a “simple” Turing-complete programming language. When executed, a Brainfuck program modifies an array of bytes via a data pointer, which is controlled by one of six commands:

- `>`, `<` increments/decrements the data pointer (`ptr++`, `ptr--`)
- `+`, `-` increments/decrements the value the data pointer points to (`(*ptr)++`, `(*ptr)--`)
- `.`, `,` writes/reads the value the data pointer points to (`putchar(*ptr)`, `*ptr = getchar()`)
- `[`, `]` form a loop that executes the inner commands as long as the value the data pointer points to is zero
`(while (*ptr == 0) { ... })`

All other characters are considered comments and are ignored.

For more details (in particular: how do I actually do something useful?!), read the Wikipedia article.

45.2 Step 1: A traditional Brainfuck VM

We first build a traditional VM for executing Brainfuck. A program for our VM is an array of instructions:

```

1 enum class op
2 {
3     ptr_inc,      // >
4     ptr_dec,      // <
5     data_inc,     // +
6     data_dec,     // -
7     write,        // .
8     read,         // ,
9     jmp_ifz,      // [, jump if zero
10    jmp,          // ], unconditional jump
11 };
12
13 template <std::size_t InstructionCapacity>
14 struct program
15 {
16     std::size_t inst_count;
17     op         inst[InstructionCapacity];
18     std::size_t inst_jmp[InstructionCapacity];
19 };

```

The first six operands directly correspond to the commands, the loop commands have been lowered to jumps. This means we don't need to scan the input string for the corresponding [or] while executing. The jump target of instruction `inst[i]` is specified in `inst_jmp[i]`; it is the index of the destination. The value of the array for a non-jump instruction is ignored.

As we ultimately want to execute a Brainfuck program known at compile-time, I'm using a simple fixed-sized array to store the instructions - we will always know an upper bound of the size.

We can now write an `execute()` function that takes a program and the `data_ptr` by using a loop and a `switch` statement:

```

1 template <std::size_t InstructionCapacity>
2 void execute(const program<InstructionCapacity>& program,
3              unsigned char* data_ptr)
4 {
5     auto inst_ptr = std::size_t(0);
6     while (inst_ptr < program.inst_count)
7     {
8         switch (program.inst[inst_ptr])
9         {
10            case op::ptr_inc:
11                ++data_ptr;
12                ++inst_ptr;
13                break;
14            case op::ptr_dec:
15                --data_ptr;

```

```

16         ++inst_ptr;
17         break;
18     case op::data_inc:
19         ++*data_ptr;
20         ++inst_ptr;
21         break;
22     case op::data_dec:
23         --*data_ptr;
24         ++inst_ptr;
25         break;
26     case op::write:
27         std::putchar(*data_ptr);
28         ++inst_ptr;
29         break;
30     case op::read:
31         *data_ptr = static_cast<unsigned char>(std::getchar());
32         ++inst_ptr;
33         break;
34     case op::jmp_ifz:
35         if (*data_ptr == 0)
36             inst_ptr = program.inst_jmp[inst_ptr];
37         else
38             ++inst_ptr;
39         break;
40     case op::jmp:
41         inst_ptr = program.inst_jmp[inst_ptr];
42         break;
43     }
44 }
45 }
```

What's left is to parse a string literal into a program. Note that we can use the length of the string literal, which is a compile-time constant, as `InstructionCapacity` (in the worst-case, each character of the string is one instruction). To implement loops, we can use a stack that remembers the position of the last opened [.

```

1 template <std::size_t N>
2 constexpr auto parse(const char (&str)[N])
3 {
4     program<N> result{};
5
6     std::size_t jump_stack[N] = {};
7     std::size_t jump_stack_top = 0;
8
9     for (auto ptr = str; *ptr; ++ptr)
```

```

10     {
11         if (*ptr == '>')
12             result.inst[result.inst_count++] = op::ptr_inc;
13         else if (*ptr == '<')
14             result.inst[result.inst_count++] = op::ptr_dec;
15         else if (*ptr == '+')
16             result.inst[result.inst_count++] = op::data_inc;
17         else if (*ptr == '-')
18             result.inst[result.inst_count++] = op::data_dec;
19         else if (*ptr == '.')
20             result.inst[result.inst_count++] = op::write;
21         else if (*ptr == ',')
22             result.inst[result.inst_count++] = op::read;
23         else if (*ptr == '[')
24         {
25             jump_stack[jump_stack_top++] = result.inst_count;
26             result.inst[result.inst_count++] = op::jmp_ifz;
27         }
28         else if (*ptr == ']')
29         {
30             auto open = jump_stack[--jump_stack_top];
31             auto close = result.inst_count++;
32
33             result.inst[close] = op::jmp;
34             result.inst_jmp[close] = open;
35
36             result.inst_jmp[open] = close + 1;
37         }
38     }
39
40     return result;
41 }
```

Putting it together, we can now parse and execute a Brainfuck program given as string literal:

```

1 // `x = std::getchar(); y = x + 3; std::putchar(y);`  

2 static constexpr auto add3 = parse(",>++<[->+<]>.");  

3  

4 // Use this array for our data_ptr.  

5 unsigned char memory[1024] = {};  

6 execute(add3, memory);
```

Note that parsing happens entirely at compile-time, but execution at runtime. It is already nice that we can do that!

The generated assembly is straightforward: clang has decided to turn the switch into a lookup table, and the code for each instruction is only a couple of assembly instructions.

If you want to go down that route more, optimizing by adding meta instructions, or JIT compilation, I highly recommend series by Eli Bendersky.

However, we're doing something different.

45.3 Step 2: Tail recursion

We're now going to change the way we write the program which doesn't really change anything, but makes it easier to motivate the next step: turning the iterative version of `execute()` with the loop into a recursive version. This is done by passing all arguments that are changed during the loops, i.e. `inst_ptr`, as additional arguments. We then remove the loop and turn `++inst_ptr; break;` into `return execute(program, memory, inst_ptr + 1)`.

Normally, recursion would be worse than iteration, as it can lead to a stack overflow. However, here we're having tail recursion, where the recursive call doesn't actually need to push a new stack frame, but can just update the arguments and jump back to the beginning of the function - just like a loop. Of course, the compiler needs to do that optimization, otherwise a loop quickly results in a stack overflow. The clang attribute `[[clang::musttail]]` can be used to force clang's hand. This is omitted in the snippet below for readability.

The new `execute()` function looks like this:

```

1 template <std::size_t InstructionCapacity>
2 void execute(const program<InstructionCapacity>& program,
3             unsigned char* data_ptr,
4             std::size_t inst_ptr = 0)
5 {
6     if (inst_ptr >= program.inst_count)
7         return; // Execution is finished.
8
9     switch (program.inst[inst_ptr])
10    {
11        case op::ptr_inc:
12            ++data_ptr;
13            return execute(program, data_ptr, inst_ptr + 1);
14        case op::ptr_dec:
15            --data_ptr;
16            return execute(program, data_ptr, inst_ptr + 1);
17        case op::data_inc:
18            **data_ptr;
19            return execute(program, data_ptr, inst_ptr + 1);
20        case op::data_dec:
21            --*data_ptr;
22            return execute(program, data_ptr, inst_ptr + 1);

```

```

23     case op::write:
24         std::putchar(*data_ptr);
25         return execute(program, data_ptr, inst_ptr + 1);
26     case op::read:
27         *data_ptr = static_cast<unsigned char>(std::getchar());
28         return execute(program, data_ptr, inst_ptr + 1);
29     case op::jmp_ifz:
30         if (*data_ptr == 0)
31             return execute(program, data_ptr, program.inst_jmp[inst_ptr]);
32         else
33             return execute(program, data_ptr, inst_ptr + 1);
34     case op::jmp:
35         return execute(program, data_ptr, program.inst_jmp[inst_ptr]);
36     }
37 }
```

Here the generated assembly appears to be slightly longer, but otherwise looks the same. This is not surprising, as we haven't really changed anything for the compiler!

In general, this strategy of implementing a VM can be better than the switch statement version: by repeatedly calling a function that takes the relevant state as arguments, the calling convention ensures that it is kept in registers! Read this blog post to learn more about the technique.

Let's actually change the generated assembly now.

45.4 Step 3: Making it a template

If you carefully look at the tail recursive version, you can make the following observation: in each recursive call, the new value of the `inst_ptr` is either given by adding a compile-time constant (1), or by reading the value from the `inst_jmp` array, which is also computed at compile-time. This means, if we know the value of `inst_ptr` before executing an instruction at compile-time, we also know its next value at compile-time. In the case of `jmp_ifz`, there is a branch on a runtime value, but the destination of each branch is fixed.

Furthermore, if we know the value of `inst_ptr` at compile-time, we also don't need to do a runtime `switch`, as the corresponding instruction in the `inst` array is also computed at compile-time.

This means, we can turn `execute(const program&, unsigned char* data_ptr, std::size_t inst_ptr)` into a template, where `program` and `inst_ptr` are given as template parameters! We can pass the program as a template parameter, as it's computed at compile-time. However, we can also pass `inst_ptr` as template parameter, as it's initially 0, and later only modified by other constants. Then we can replace the `switch` by `if constexpr`, and instead of tail recursion, we have tail calls to a different instantiation of the template.

```

1 template <const auto& Program, std::size_t InstPtr = 0>
2 constexpr void execute(unsigned char* data_ptr)
3 {
```

```
4     if constexpr (InstPtr >= Program.inst_count)
5     {
6         // Execution is finished.
7         return;
8     }
9     else if constexpr (Program.inst[InstPtr] == op::ptr_inc)
10    {
11        ++data_ptr;
12        return execute<Program, InstPtr + 1>(data_ptr);
13    }
14    else if constexpr (Program.inst[InstPtr] == op::ptr_dec)
15    {
16        --data_ptr;
17        return execute<Program, InstPtr + 1>(data_ptr);
18    }
19    else if constexpr (Program.inst[InstPtr] == op::data_inc)
20    {
21        ++*data_ptr;
22        return execute<Program, InstPtr + 1>(data_ptr);
23    }
24    else if constexpr (Program.inst[InstPtr] == op::data_dec)
25    {
26        --*data_ptr;
27        return execute<Program, InstPtr + 1>(data_ptr);
28    }
29    else if constexpr (Program.inst[InstPtr] == op::write)
30    {
31        std::putchar(*data_ptr);
32        return execute<Program, InstPtr + 1>(data_ptr);
33    }
34    else if constexpr (Program.inst[InstPtr] == op::read)
35    {
36        *data_ptr = static_cast<char>(std::getchar());
37        return execute<Program, InstPtr + 1>(data_ptr);
38    }
39    else if constexpr (Program.inst[InstPtr] == op::jmp_ifz)
40    {
41        if (*data_ptr == 0)
42            return execute<Program, Program.inst_jmp[InstPtr]>(data_ptr);
43        else
44            return execute<Program, InstPtr + 1>(data_ptr);
45    }
```

```

46     else if constexpr (Program.inst[InstPtr] == op::jmp)
47     {
48         return execute<Program, Program.inst_jmp[InstPtr]>(data_ptr);
49     }
50 }
```

Note that we don't even need C++20's extended NTTP to pass the program as a template argument. As it's a global variable, we can pass it by reference. This also means that name mangling is a lot nicer, as it's just an address, and doesn't need to the value of every array member.

Now take a look at the assembly: all the dispatch has disappeared, and it is replaced by “call `std::getchar()`, add 3, call `std::putchar()`! This is possible, because we're doing the dispatch entirely at compile-time, the compiler sees a series of tail calls, which are trivial to fuse together and optimize.

Furthermore, as all accesses into `Program`'s arrays are compile-time constants, there is no need for `Program` to appear in the binary at all. This means that there is no additional memory to store the instructions.

45.5 Conclusion

While nice, how is this actually useful? We can just write the equivalent behavior in C++ directly, without bothering to parse a Brainfuck program.

However, there are situations where you want to specify something in a different language at compile-time, and have it execute at compile time. For example, regexes: given a compile-time string literal, we can generate instructions for our regex VM, and leverage this technique to get efficient code generation for runtime execution. This is basically the way Hana's CTRE library works. Similarly, I'm currently using it in lex to generate efficient code for matching against a set of string literals.

Whenever you want to specify something in a DSL at compile-time and have it execute efficiently on dynamic input, this hybrid approach of separating the static program state and the dynamic data, using `if constexpr` and tail recursion (or just relying on inlining if you don't have loops) works.

If you've liked this blog post, consider donating or otherwise supporting me.

Chapter 46

Improving the State of Debug Performance in C++

● Cameron DaCamara 📅 2022-12-13 🔍 ★★★

In this blog we will explore one change the MSVC compiler has implemented in an effort to improve the codegen quality of applications in debug mode. We will highlight what the change does, and how it could be extended for the future. If debug performance is something you care about for your C++ projects, then Visual Studio 2022 version 17.5 is making that experience even better!

Please note that this blog will contain some assembly but being an expert in assembly is not required.

46.1 Overview

- Motivation: why we care about debugging performance.
- Show me some code!: A few simple examples of before and after.
- How we did it: About our new intrinsic and how you could use it.
- Looking ahead: What else we’re doing to make the experience better.

46.2 Motivation

You might notice that the title of this blog is a play on words based on a recent popular blog post of a similar name, “the sad state of debug performance in c++” . In the blog Vittorio Romeo highlights some general C++ shortcomings when it comes to debugging performance. Vittorio also also filed this Developer Community ticket “‘std::move‘ (and similar functions) result in poor debug performance and worse debugging experience”; thanks to him and everyone who voted! Much of the reason for the observed slowdown is the cost of abstraction, with the notable example of `std::move` where the following code:

```
int i = 0;  
std::move(i);
```

Would generate a function call when the code is conceptually:

```
int i = 0;
static_cast<int&&>(i);
```

The function `std::move` is conceptually a named cast, much like `static_cast` but with a contextual meaning for code around it. The penalty for using this named cast is that you get a function call generated in the debug assembly. Here's the assembly of the two examples above:

<code>std::move</code> (click to expand)	<code>static_cast</code> (click to expand)
<pre>main PROC sub rsp, 56 ; 00000038H mov DWORD PTR i\$[rsp], 0 lea rcx, QWORD PTR i\$[rsp] call ??\$move@AEAH@std@@YASSQEAHAEAH@Z xor eax, eax add rsp, 56 ; 00000038H ret 0 main ENDP</pre>	<pre>main PROC sub rsp, 24 mov DWORD PTR i\$[rsp], 0 xor eax, eax add rsp, 24 ret 0 main ENDP</pre>

Note to readers: All code samples compiled in this blog were compiled with “/Od /std:c++latest”

On the surface, the compiler only generated 2 extra instructions in the `std::move` case, but the 'call' instruction, in particular, is both expensive and executes this code in addition to the code above:

```
1 ??\?$move@AEAH@std@@YA\$\$QEAHAEAH@Z PROC??I??I??I; std::move<int &>, COMDAT
2 mov??IQWORD PTR [rsp+8], rcx
3 mov??Irax, QWORD PTR _Arg\$\$[rsp]
4 ret??IO
5 ??\?$move@AEAH@std@@YA\$\$QEAHAEAH@Z ENDP??I??I??I; std::move<int &>
```

Note: to generate the assembly above, the compiler can be provided with the /Fa option. Furthermore, the weird names like “??\$move@AEAH@std@@@YASSQEAHAEAH@Z” are a mangled name of the function template specialization of `std::move`.

So really your binary is now at a 5 instruction deficit to the `static_cast` code, and this cost is multiplied by the number of times that `std::move` is used.

Some compilers have already implemented some mechanism to acknowledge meta functions like `std::move` and `std::forward` as compiler intrinsics (as noted in Vittorio's blog) and this support is done completely in the compiler front-end. As of 17.5, MSVC is offering better debugging performance by acknowledging these meta functions as well! More on how we do it later in this blog, but first...

46.3 Show me some code!

Note to readers: to take advantage of the new codegen quality, you will need to provide the /permissive- compiler option. Also worthy to note that /permissive- is implied when /std:c++20 or /std:c++latest is used.

Let's take the simple example above again and make it a full program:

```
#include <utility>

int main() {
    int i = 0;
    std::move(i);
    std::forward<int&>(i);
}
```

Here's the generated assembly difference between 17.4 and 17.5:

17.4 (click to expand)	17.5 (click to expand)
<pre>_Arg\$ = 8 ??\$forward@AEAH@std@@YAAEAHAEAH@Z PROC mov QWORD PTR [rsp+8], rcx mov rax, QWORD PTR _Arg\$[rsp] ret 0 ??\$forward@AEAH@std@@YAAEAHAEAH@Z ENDP _TEXT ENDS _TEXT SEGMENT _Arg\$ = 8 ??\$move@AEAH@std@@YA\$\$QEAHAEAH@Z PROC mov QWORD PTR [rsp+8], rcx mov rax, QWORD PTR _Arg\$[rsp] ret 0 ??\$move@AEAH@std@@YA\$\$QEAHAEAH@Z ENDP _TEXT ENDS _TEXT SEGMENT i\$ = 32 main PROC sub rsp, 56 ; 00000038H mov DWORD PTR i\$[rsp], 0 lea rcx, QWORD PTR i\$[rsp] call ??\$move@AEAH@std@@YA\$\$QEAHAEAH@Z lea rcx, QWORD PTR i\$[rsp] call ??\$forward@AEAH@std@@YAAEAHAEAH@Z xor eax, eax add rsp, 56 ; 00000038H ret 0 main ENDP</pre>	<pre>i\$ = 0 main PROC \$LN3: sub rsp, 24 mov DWORD PTR i\$[rsp], 0 xor eax, eax add rsp, 24 ret 0 main ENDP</pre>

Assembly reading tip: The **main PROC** above is our **main** function in the C++ code. The instructions that follow **main PROC** are what your CPU will execute when your program is first invoked. In the case above, it is clear that the code produced by 17.5 is much smaller, which can sometimes be an indication of a performance win. For the purposes of this blog, the performance win is both in the size of the code produced and the reduction in indirections due to inlining the 'call' instruction to **std::move** and **std::forward**. For the purposes of this blog we will rely on the newly generated assembly reduced complexity as an indicator of possible performance wins.

Yes, you read that right, the generated code in 17.5 doesn't even create assembly entries for **std::move** or **std::forward**—which makes sense, they're never called.

Let's look at a slightly more complicated code example:

```

1 #include <utility>
2
3 template <typename T>
4 void add_1_impl(T&& x) {
5     std::forward<T>(x) += std::move(1);
6 }
7
8 template <typename T, int N>
9 void add_1(T (&arr)[N]) {
10    for (auto&& e : arr) {
11        add_1_impl(e);
12    }
13 }
14
15 int main() {
16     int arr[10]{};
17     add_1(arr);
18 }
```

In this code all we want to do is add 1 to all elements of the array. Here's the table (only showing the `add_1_impl` function with `std::forward` and `std::move`):

17.4 (click to expand)	17.5 (click to expand)
<pre>??\$add_1_impl@AEAH@@YAXAEAH@Z PROC \$LN3: mov QWORD PTR [rsp+8], rcx sub rsp, 72 ; 00000048H mov DWORD PTR \$T1[rsp], 1 lea rcx, QWORD PTR \$T1[rsp] call ??\$move@H@std@@YASSQEAH\$\$QEAH@Z mov eax, DWORD PTR [rax] mov DWORD PTR tv72[rsp], eax mov rcx, QWORD PTR x\$[rsp] call ??\$forward@AEAH@std@@YAAEAAHAEAH@Z mov QWORD PTR tv68[rsp], rax mov rax, QWORD PTR tv68[rsp] mov eax, DWORD PTR [rax] mov DWORD PTR tv70[rsp], eax mov eax, DWORD PTR tv72[rsp] mov ecx, DWORD PTR tv70[rsp] add ecx, eax mov eax, ecx mov rcx, QWORD PTR tv68[rsp] mov DWORD PTR [rcx], eax add rsp, 72 ; 00000048H ret 0 ??\$add_1_impl@AEAH@@YAXAEAH@Z ENDP</pre>	<pre>??\$add_1_impl@AEAH@@YAXAEAH@Z PROC \$LN3: mov QWORD PTR [rsp+8], rcx sub rsp, 24 mov DWORD PTR \$T1[rsp], 1 mov rax, QWORD PTR x\$[rsp] mov eax, DWORD PTR [rax] add eax, DWORD PTR \$T1[rsp] mov rcx, QWORD PTR x\$[rsp] mov DWORD PTR [rcx], eax add rsp, 24 ret 0 ??\$add_1_impl@AEAH@@YAXAEAH@Z ENDP</pre>

17.4 has 21 instructions while 17.5 has only 10, but this comparison is made that much more extreme by the fact that we are calling `add_1_impl` in a loop so the complexity of executed instructions in 17.4 can

ostensibly be significantly more costly than in 17.5—worse than that, actually, because we’re not accounting for the instructions executed in the functions `std::forward` and `std::move`.

Let’s make the code sample even more interesting and extreme to illustrate the visible differences. It might be observed that if we manually unroll the loop above we can get a performance win, so let’s do that using templates:

```

1 #include <utility>
2
3 template <typename T, int N, std::size_t... Is>
4 void add_1_impl(std::index_sequence<Is...>, T (&arr)[N]) {
5     ((std::forward<T&>(arr[Is]) += std::move(1)), ...);
6 }
7
8 template <typename T, int N>
9 void add_1(T (&arr)[N]) {
10     add_1_impl(std::make_index_sequence<N>{}, arr);
11 }
12
13 int main() {
14     int arr[10]{};
15     add_1(arr);
16 }
```

The code above replaces the loop in the previous example with a single fold expression. Let’s peek at the codegen (again only snipping `add_1_impl` with `std::forward` and `std::move`, we also replace the mangled function name with `add_1_impl<...>`):

- 17.4

```

1     add_1_impl<...> PROC
2     \$LN3:
3         mov^__IQWORD PTR [rsp+16], rdx
4         mov^__IBYTE PTR [rsp+8], cl
5         sub^__Irsp, 248^__I; 000000f8H
6         mov^__IDWORD PTR \$T1[rsp], 1
7         lea^__Ircx, QWORD PTR \$T1[rsp]
8         call^__I??\$move@H@std@@YA\$QEAH\$QEAH@Z
9         mov^__Ieax, DWORD PTR [rax]
10        mov^__IDWORD PTR tv74[rsp], eax
11        mov^__Ieax, 4
12        imul^__Irax, rax, 0
13        mov^__Ircx, QWORD PTR arr\[$rsp]
14        add^__Ircx, rax
15        mov^__Irax, rcx
16        mov^__Ircx, rax
```

```
17  call ^?`$forward@AEAH@std@@YAAEAHAEAH@Z
18  mov ^?IQWORD PTR tv70[rsp], rax
19  mov ^?Irax, QWORD PTR tv70[rsp]
20  mov ^?Ieax, DWORD PTR [rax]
21  mov ^?IDWORD PTR tv72[rsp], eax
22  mov ^?Ieax, DWORD PTR tv74[rsp]
23  mov ^?Iecx, DWORD PTR tv72[rsp]
24  add ^?Iecx, eax
25  mov ^?Ieax, ecx
26  mov ^?Ircx, QWORD PTR tv70[rsp]
27  mov ^?IDWORD PTR [rcx], eax
28  mov ^?IDWORD PTR $T2[rsp], 1
29  lea ^?Ircx, QWORD PTR $T2[rsp]
30  call ^?`$move@H@std@@YA$QEAH$QEAH@Z
31  mov ^?Ieax, DWORD PTR [rax]
32  mov ^?IDWORD PTR tv86[rsp], eax
33  mov ^?Ieax, 4
34  imul ^?Irax, rax, 1
35  mov ^?Ircx, QWORD PTR arr[$][rsp]
36  add ^?Ircx, rax
37  mov ^?Irax, rcx
38  mov ^?Ircx, rax
39  call ^?`$forward@AEAH@std@@YAAEAHAEAH@Z
40  mov ^?IQWORD PTR tv82[rsp], rax
41  mov ^?Irax, QWORD PTR tv82[rsp]
42  mov ^?Ieax, DWORD PTR [rax]
43  mov ^?IDWORD PTR tv84[rsp], eax
44  mov ^?Ieax, DWORD PTR tv86[rsp]
45  mov ^?Iecx, DWORD PTR tv84[rsp]
46  add ^?Iecx, eax
47  mov ^?Ieax, ecx
48  mov ^?Ircx, QWORD PTR tv82[rsp]
49  mov ^?IDWORD PTR [rcx], eax
50  mov ^?IDWORD PTR $T3[rsp], 1
51  lea ^?Ircx, QWORD PTR $T3[rsp]
52  call ^?`$move@H@std@@YA$QEAH$QEAH@Z
53  mov ^?Ieax, DWORD PTR [rax]
54  mov ^?IDWORD PTR tv130[rsp], eax
55  mov ^?Ieax, 4
56  imul ^?Irax, rax, 2
57  mov ^?Ircx, QWORD PTR arr[$][rsp]
58  add ^?Ircx, rax
```

```
59    mov ^ Irax, rcx
60    mov ^ Ircx, rx
61    call ^ I??\forward@AEAH@std@@YAAEAHAEAH@Z
62    mov ^ IQWORD PTR tv94[rsp], rx
63    mov ^ Irax, QWORD PTR tv94[rsp]
64    mov ^ Ieax, DWORD PTR [rax]
65    mov ^ IDWORD PTR tv128[rsp], eax
66    mov ^ Ieax, DWORD PTR tv130[rsp]
67    mov ^ Iecx, DWORD PTR tv128[rsp]
68    add ^ Iecx, eax
69    mov ^ Ieax, ecx
70    mov ^ Ircx, QWORD PTR tv94[rsp]
71    mov ^ IDWORD PTR [rcx], eax
72    mov ^ IDWORD PTR \$T4[rsp], 1
73    lea ^ Ircx, QWORD PTR \$T4[rsp]
74    call ^ I??\move@H@std@@YA\$QEAH\$QEAH@Z
75    mov ^ Ieax, DWORD PTR [rax]
76    mov ^ IDWORD PTR tv142[rsp], eax
77    mov ^ Ieax, 4
78    imul ^ Irax, rx, 3
79    mov ^ Ircx, QWORD PTR arr\$[rsp]
80    add ^ Ircx, rx
81    mov ^ Irax, rcx
82    mov ^ Ircx, rx
83    call ^ I??\forward@AEAH@std@@YAAEAHAEAH@Z
84    mov ^ IQWORD PTR tv138[rsp], rx
85    mov ^ Irax, QWORD PTR tv138[rsp]
86    mov ^ Ieax, DWORD PTR [rax]
87    mov ^ IDWORD PTR tv140[rsp], eax
88    mov ^ Ieax, DWORD PTR tv142[rsp]
89    mov ^ Iecx, DWORD PTR tv140[rsp]
90    add ^ Iecx, eax
91    mov ^ Ieax, ecx
92    mov ^ Ircx, QWORD PTR tv138[rsp]
93    mov ^ IDWORD PTR [rcx], eax
94    mov ^ IDWORD PTR \$T5[rsp], 1
95    lea ^ Ircx, QWORD PTR \$T5[rsp]
96    call ^ I??\move@H@std@@YA\$QEAH\$QEAH@Z
97    mov ^ Ieax, DWORD PTR [rax]
98    mov ^ IDWORD PTR tv154[rsp], eax
99    mov ^ Ieax, 4
100   imul ^ Irax, rx, 4
```

```
101 mov^__Ircx, QWORD PTR arr[$[rsp]
102 add^__Ircx, rax
103 mov^__Irax, rcx
104 mov^__Ircx, rax
105 call^__I??\forward@AEAH@std@@YAAEAHAEAH@Z
106 mov^__IQWORD PTR tv150[rsp], rax
107 mov^__Irax, QWORD PTR tv150[rsp]
108 mov^__Ieax, DWORD PTR [rax]
109 mov^__IDWORD PTR tv152[rsp], eax
110 mov^__Ieax, DWORD PTR tv154[rsp]
111 mov^__Iecx, DWORD PTR tv152[rsp]
112 add^__Iecx, eax
113 mov^__Ieax, ecx
114 mov^__Ircx, QWORD PTR tv150[rsp]
115 mov^__IDWORD PTR [rcx], eax
116 mov^__IDWORD PTR $T6[rsp], 1
117 lea^__Ircx, QWORD PTR $T6[rsp]
118 call^__I??\move@H@std@@YA$QEAH$QEAH@Z
119 mov^__Ieax, DWORD PTR [rax]
120 mov^__IDWORD PTR tv166[rsp], eax
121 mov^__Ieax, 4
122 imul^__Irax, rax, 5
123 mov^__Ircx, QWORD PTR arr[$[rsp]
124 add^__Ircx, rax
125 mov^__Irax, rcx
126 mov^__Ircx, rax
127 call^__I??\forward@AEAH@std@@YAAEAHAEAH@Z
128 mov^__IQWORD PTR tv162[rsp], rax
129 mov^__Irax, QWORD PTR tv162[rsp]
130 mov^__Ieax, DWORD PTR [rax]
131 mov^__IDWORD PTR tv164[rsp], eax
132 mov^__Ieax, DWORD PTR tv166[rsp]
133 mov^__Iecx, DWORD PTR tv164[rsp]
134 add^__Iecx, eax
135 mov^__Ieax, ecx
136 mov^__Ircx, QWORD PTR tv162[rsp]
137 mov^__IDWORD PTR [rcx], eax
138 mov^__IDWORD PTR $T7[rsp], 1
139 lea^__Ircx, QWORD PTR $T7[rsp]
140 call^__I??\move@H@std@@YA$QEAH$QEAH@Z
141 mov^__Ieax, DWORD PTR [rax]
142 mov^__IDWORD PTR tv178[rsp], eax
```

```
143     mov ^Ieax, 4
144     imul ^Irax, rax, 6
145     mov ^Ircx, QWORD PTR arr[$rsp]
146     add ^Ircx, rax
147     mov ^Irax, rcx
148     mov ^Ircx, rax
149     call ^I??\$forward@AEAH@std@@YAAEAHAEAH@Z
150     mov ^IQWORD PTR tv174[rsp], rax
151     mov ^Irax, QWORD PTR tv174[rsp]
152     mov ^Ieax, DWORD PTR [rax]
153     mov ^IDWORD PTR tv176[rsp], eax
154     mov ^Ieax, DWORD PTR tv178[rsp]
155     mov ^Iecx, DWORD PTR tv176[rsp]
156     add ^Iecx, eax
157     mov ^Ieax, ecx
158     mov ^Ircx, QWORD PTR tv174[rsp]
159     mov ^IDWORD PTR [rcx], eax
160     mov ^IDWORD PTR $T8[rsp], 1
161     lea ^Ircx, QWORD PTR $T8[rsp]
162     call ^I??\$move@H@std@@YA\$\$QEAH\$\$QEAH@Z
163     mov ^Ieax, DWORD PTR [rax]
164     mov ^IDWORD PTR tv190[rsp], eax
165     mov ^Ieax, 4
166     imul ^Irax, rax, 7
167     mov ^Ircx, QWORD PTR arr[$rsp]
168     add ^Ircx, rax
169     mov ^Irax, rcx
170     mov ^Ircx, rax
171     call ^I??\$forward@AEAH@std@@YAAEAHAEAH@Z
172     mov ^IQWORD PTR tv186[rsp], rax
173     mov ^Irax, QWORD PTR tv186[rsp]
174     mov ^Ieax, DWORD PTR [rax]
175     mov ^IDWORD PTR tv188[rsp], eax
176     mov ^Ieax, DWORD PTR tv190[rsp]
177     mov ^Iecx, DWORD PTR tv188[rsp]
178     add ^Iecx, eax
179     mov ^Ieax, ecx
180     mov ^Ircx, QWORD PTR tv186[rsp]
181     mov ^IDWORD PTR [rcx], eax
182     mov ^IDWORD PTR $T9[rsp], 1
183     lea ^Ircx, QWORD PTR $T9[rsp]
184     call ^I??\$move@H@std@@YA\$\$QEAH\$\$QEAH@Z
```

```
185 mov^~Ieax, DWORD PTR [rax]
186 mov^~IDWORD PTR tv202[rsp], eax
187 mov^~Ieax, 4
188 imul^~Irax, rax, 8
189 mov^~Ircx, QWORD PTR arr\[rsp]
190 add^~Ircx, rax
191 mov^~Irax, rcx
192 mov^~Ircx, rax
193 call^~I??\$forward@AEAH@std@@YAAEAHAEAH@Z
194 mov^~IQWORD PTR tv198[rsp], rax
195 mov^~Irax, QWORD PTR tv198[rsp]
196 mov^~Ieax, DWORD PTR [rax]
197 mov^~IDWORD PTR tv200[rsp], eax
198 mov^~Ieax, DWORD PTR tv202[rsp]
199 mov^~Iecx, DWORD PTR tv200[rsp]
200 add^~Iecx, eax
201 mov^~Ieax, ecx
202 mov^~Ircx, QWORD PTR tv198[rsp]
203 mov^~IDWORD PTR [rcx], eax
204 mov^~IDWORD PTR \$T10[rsp], 1
205 lea^~Ircx, QWORD PTR \$T10[rsp]
206 call^~I??\$move@H@std@@YA\$\$QEAH\$\$QEAH@Z
207 mov^~Ieax, DWORD PTR [rax]
208 mov^~IDWORD PTR tv214[rsp], eax
209 mov^~Ieax, 4
210 imul^~Irax, rax, 9
211 mov^~Ircx, QWORD PTR arr\[rsp]
212 add^~Ircx, rax
213 mov^~Irax, rcx
214 mov^~Ircx, rax
215 call^~I??\$forward@AEAH@std@@YAAEAHAEAH@Z
216 mov^~IQWORD PTR tv210[rsp], rax
217 mov^~Irax, QWORD PTR tv210[rsp]
218 mov^~Ieax, DWORD PTR [rax]
219 mov^~IDWORD PTR tv212[rsp], eax
220 mov^~Ieax, DWORD PTR tv214[rsp]
221 mov^~Iecx, DWORD PTR tv212[rsp]
222 add^~Iecx, eax
223 mov^~Ieax, ecx
224 mov^~Ircx, QWORD PTR tv210[rsp]
225 mov^~IDWORD PTR [rcx], eax
226 add^~Irsp, 248^~I; 000000f8H
```

```

227     ret ~~ IO
228     add_1_impl<...> ENDP

```

- 17.5

```

1      add_1_impl<...> PROC
2      \$LN3:
3      mov ~~ IQWORD PTR [rsp+16], rdx
4      mov ~~ IBYTE PTR [rsp+8], cl
5      sub ~~ Irsp, 56 ~~ I; 00000038H
6      mov ~~ IDWORD PTR \$T1[rsp], 1
7      mov ~~ Ieax, 4
8      imul ~~ Irax, rax, 0
9      mov ~~ Ircx, QWORD PTR arr\[$rsp]
10     mov ~~ Ieax, DWORD PTR [rcx+rax]
11     add ~~ Ieax, DWORD PTR \$T1[rsp]
12     mov ~~ Iecx, 4
13     imul ~~ Ircx, rcx, 0
14     mov ~~ Irdx, QWORD PTR arr\[$rsp]
15     mov ~~ IDWORD PTR [rdx+rcx], eax
16     mov ~~ IDWORD PTR \$T2[rsp], 1
17     mov ~~ Ieax, 4
18     imul ~~ Irax, rax, 1
19     mov ~~ Ircx, QWORD PTR arr\[$rsp]
20     mov ~~ Ieax, DWORD PTR [rcx+rax]
21     add ~~ Ieax, DWORD PTR \$T2[rsp]
22     mov ~~ Iecx, 4
23     imul ~~ Ircx, rcx, 1
24     mov ~~ Irdx, QWORD PTR arr\[$rsp]
25     mov ~~ IDWORD PTR [rdx+rcx], eax
26     mov ~~ IDWORD PTR \$T3[rsp], 1
27     mov ~~ Ieax, 4
28     imul ~~ Irax, rax, 2
29     mov ~~ Ircx, QWORD PTR arr\[$rsp]
30     mov ~~ Ieax, DWORD PTR [rcx+rax]
31     add ~~ Ieax, DWORD PTR \$T3[rsp]
32     mov ~~ Iecx, 4
33     imul ~~ Ircx, rcx, 2
34     mov ~~ Irdx, QWORD PTR arr\[$rsp]
35     mov ~~ IDWORD PTR [rdx+rcx], eax
36     mov ~~ IDWORD PTR \$T4[rsp], 1
37     mov ~~ Ieax, 4
38     imul ~~ Irax, rax, 3
39     mov ~~ Ircx, QWORD PTR arr\[$rsp]

```

```
40 mov^__Ieax, DWORD PTR [rcx+rax]
41 add^__Ieax, DWORD PTR \$T4[rsp]
42 mov^__Iecx, 4
43 imul^__Ircx, rcx, 3
44 mov^__Irdx, QWORD PTR arr\[$rsp]
45 mov^__IDWORD PTR [rdx+rcx], eax
46 mov^__IDWORD PTR \$T5[rsp], 1
47 mov^__Ieax, 4
48 imul^__Irax, rax, 4
49 mov^__Ircx, QWORD PTR arr\[$rsp]
50 mov^__Ieax, DWORD PTR [rcx+rax]
51 add^__Ieax, DWORD PTR \$T5[rsp]
52 mov^__Iecx, 4
53 imul^__Ircx, rcx, 4
54 mov^__Irdx, QWORD PTR arr\[$rsp]
55 mov^__IDWORD PTR [rdx+rcx], eax
56 mov^__IDWORD PTR \$T6[rsp], 1
57 mov^__Ieax, 4
58 imul^__Irax, rax, 5
59 mov^__Ircx, QWORD PTR arr\[$rsp]
60 mov^__Ieax, DWORD PTR [rcx+rax]
61 add^__Ieax, DWORD PTR \$T6[rsp]
62 mov^__Iecx, 4
63 imul^__Ircx, rcx, 5
64 mov^__Irdx, QWORD PTR arr\[$rsp]
65 mov^__IDWORD PTR [rdx+rcx], eax
66 mov^__IDWORD PTR \$T7[rsp], 1
67 mov^__Ieax, 4
68 imul^__Irax, rax, 6
69 mov^__Ircx, QWORD PTR arr\[$rsp]
70 mov^__Ieax, DWORD PTR [rcx+rax]
71 add^__Ieax, DWORD PTR \$T7[rsp]
72 mov^__Iecx, 4
73 imul^__Ircx, rcx, 6
74 mov^__Irdx, QWORD PTR arr\[$rsp]
75 mov^__IDWORD PTR [rdx+rcx], eax
76 mov^__IDWORD PTR \$T8[rsp], 1
77 mov^__Ieax, 4
78 imul^__Irax, rax, 7
79 mov^__Ircx, QWORD PTR arr\[$rsp]
80 mov^__Ieax, DWORD PTR [rcx+rax]
81 add^__Ieax, DWORD PTR \$T8[rsp]
```

```

82      mov~~ Iecx, 4
83      imul~~ Ircx, rcx, 7
84      mov~~ Irdx, QWORD PTR arr\$[rsp]
85      mov~~ IDWORD PTR [rdx+rcx], eax
86      mov~~ IDWORD PTR \${T9}[rsp], 1
87      mov~~ Ieax, 4
88      imul~~ Irax, rax, 8
89      mov~~ Ircx, QWORD PTR arr\$[rsp]
90      mov~~ Ieax, DWORD PTR [rcx+rax]
91      add~~ Ieax, DWORD PTR \${T9}[rsp]
92      mov~~ Iecx, 4
93      imul~~ Ircx, rcx, 8
94      mov~~ Irdx, QWORD PTR arr\$[rsp]
95      mov~~ IDWORD PTR [rdx+rcx], eax
96      mov~~ IDWORD PTR \${T10}[rsp], 1
97      mov~~ Ieax, 4
98      imul~~ Irax, rax, 9
99      mov~~ Ircx, QWORD PTR arr\$[rsp]
100     mov~~ Ieax, DWORD PTR [rcx+rax]
101     add~~ Ieax, DWORD PTR \${T10}[rsp]
102     mov~~ Iecx, 4
103     imul~~ Ircx, rcx, 9
104     mov~~ Irdx, QWORD PTR arr\$[rsp]
105     mov~~ IDWORD PTR [rdx+rcx], eax
106     add~~ Irsp, 56~~I; 00000038H
107     ret~~ I0
108     add_1_impl<...> ENDP

```

Our 17.4 example clocks in at a whopping 226 instructions while our 17.5 example is only 106 and the complexity of the instructions in 17.4 appears to be far more costly due to the number of call frame setups and 'call' instructions which are not present on the 17.5 side.

OK, perhaps the examples above are contrived and it might be far-fetched to think that code like the above would truly impact performance, but let's take some code that is all but guaranteed to have some kind of real world application:

```
#include <vector>

int main() {
    std::vector<int> v;
    v.push_back(1);
}
```

I will save you the massive assembly output on this one and simply callout the assembly size difference:

- 17.4: 3136

- 17.5: 3063

Your assembly is 74 instructions shorter just by the compiler eliding these meta functions, and you can all but guarantee that in the places where `std::move` and `std::forward` are used, they may be used in a loop (i.e. resizing the vector and moving the elements to a new memory block). Furthermore, since these meta functions are never instantiated the corresponding .obj, .lib, and .pdb will be slightly smaller after upgrading to 17.5.

46.4 How we did it

Rather than try to make the compiler aware of meta functions that act as named, no-op casts (i.e. the cast does not require a pointer adjustment), the compiler took an alternative approach and implemented this new inlining ability using a C++ attribute: `[[msvc::intrinsic]]`.

The new attribute will semantically replace a function call with a cast to that function's return type if the function definition is decorated with `[[msvc::intrinsic]]`. You can see how we applied this new attribute in the STL: GH3182. The reason the compiler decided to go down the attribute route is that we want to eventually extend the scenarios it can cover and offer a data-driven approach to selectively decorate code with the new functionality. The latter is important for users of MSVC as well.

You can read more about the attribute and its constraints and semantics in the Microsoft-specific attributes section of our documentation.

46.5 Looking ahead…

The compiler front-end is not alone in this story of improving the performance of generated code for debugging purposes, the compiler back-end is also working very hard on some debug codegen scenarios that they will share in the coming months.

Call to action: what types of debugging optimizations matter to you? What optimizations for debug code would you like to see MSVC implement?

Especially if you work for a game studio, please help us find out what your debugging workflow looks like by taking this survey: <https://aka.ms/MSVCDebugSurvey>. Data like this helps the team focus on what workflows are important to you.

Onward and upward!

46.5.1 Closing

As always, we welcome your feedback. Feel free to send any comments through e-mail at `visualcpp@microsoft.com` or through Twitter `@visualc`. Also, feel free to follow Cameron DaCamara on Twitter `@starfreakclone`.

If you encounter other problems with MSVC in VS 2019/2022 please let us know via the Report a Problem option, either from the installer or the Visual Studio IDE itself. For suggestions or bug reports, let us know through DevComm.

Chapter 47

Simplify Code with `if constexpr` and Concepts in C++17/C++20

👤 Bartłomiej Filipek 📅 2022-8-8 | 🏷★★★

Before C++17, we had a few quite ugly-looking ways to write static if (if that works at compile time). For example, you could use tag dispatching or SFINAE. Fortunately, that's changed, and we can now benefit from `if constexpr` and concepts from C++20!

Let's see how we can use it and replace some `std::enable_if` code.

- Updated in April 2021: C++20 changes - concepts.
- Updated in August 2022: More `if constexpr` examples (use case 4).

47.1 Intro

Compile-time if in the form of `if constexpr` is a fantastic feature that went into C++17. With this functionality, we can improve the readability of some heavily templated code.

Additionally, with C++20, we got Concepts! This is another step to having almost “natural” compile-time code.

This blog post was inspired by an article @Meeting C++ with a similar title. I've found four additional examples that can illustrate this new feature:

- Number comparisons
- (New!) Computing average on a container
- Factories with a variable number of arguments
- Examples of some actual production code

But to start, I'd like to recall the basic knowledge about `enable_if` to set some background.

47.2 Why compile-time if?

Let's start with an example that tries to convert an input into a string:

```

1 #include <string>
2 #include <iostream>
3
4 template <typename T>
5 std::string str(T t) {
6     return std::to_string(t);
7 }
8
9 std::string str(const std::string& s) {
10    return s;
11 }
12
13 std::string str(const char* s) {
14    return s;
15 }
16
17 std::string str(bool b) {
18    return b ? "true" : "false";
19 }
20
21 int main() {
22     std::cout << str("hello") << '\n';
23     std::cout << str(std::string{"hi!"}) << '\n';
24     std::cout << str(42) << '\n';
25     std::cout << str(42.2) << '\n';
26     std::cout << str(true) << '\n';
27 }
```

Run at Compiler Explorer.

As you can see, there are three function overloads for concrete types and one function template for all other types that should support `to_string()`. This seems to work, but can we convert that into a single function?

Can the “normal” `if` just work?

Here's a test code:

```

template <typename T>
std::string str(T t) {
    if (std::is_convertible_v<T, std::string>)
        return t;
    else if (std::is_same_v<T, bool>)
```

```

        return t ? "true" : "false";
    else
        return std::to_string(t);
}

```

It sounds simple...but try to compile this code:

```
// code that calls our function
auto t = str("10"s);
```

You might get something like this:

```
In instantiation of 'std::__cxx11::string str(T) [with T =
std::__cxx11::basic_string<char>; std::__cxx11::string =
std::__cxx11::basic_string<char>]':
required from here
error: no matching function for call to
'to_string(std::__cxx11::basic_string<char>&)'
    return std::to_string(t);
```

`is_convertible` yields true for the type we used (`std::string`), and we can just return `t` without any conversion...so what's wrong?

Here's the main point:

The compiler compiled all branches and found an error in the `else` case. It couldn't reject the "invalid" code for this particular template instantiation.

That's why we need `static_if` that would "discard" code and compile only the matching statement. To be precise, we'd like to have a syntax check for the whole code, but some parts of the routine would not be instantiated.

47.3 std::enable_if

One way to write `static_if` in C++11/14 is to use `enable_if`.

`enable_if` (and `enable_if_v` since C++14). It has quite a strange syntax:

```
template< bool B, class T = void >
struct enable_if;
```

`enable_if` will evaluate to `T` if the input condition `B` is true. Otherwise, it's SFINAE, and a particular function overload is removed from the overload set. This means that on false the compiler "rejects" the code - this is precisely what we need.

We can rewrite our basic example to:

```
1 template <typename T>
2 enable_if_t<is_convertible_v<T, string>, string> strOld(T t) {
3     return t;
4 }
```

```

5
6 template <typename T>
7 enable_if_t<!is_convertible_v<T, string>, string> strOld(T t) {
8     return to_string(t);
9 }
10 // std:: prefix ommited

```

Not easy...right? Additionally, this version looks far more complicated than the separate functions and regular function overloading we had at the start.

That's why we need **if constexpr** from C++17 that can help in such cases.

After you read the post, you'll be able to rewrite our **str** utility quickly (or find the solution at the end of this post).

To understand the new feature, let's start with some basic cases:

47.3.1 Use Case 1 - Comparing Numbers

First, let's start with a simple example: **close_enough** function that works on two numbers. If the numbers are not floating points (like when we have two **ints**), we can compare them directly. Otherwise, for floating points, it's better to use some **abs < epsilon** checks.

I've found this sample from at Practical Modern C++ Teaser - a fantastic walkthrough of modern C++ features by Patrice Roy. He was also very kind and allowed me to include this example.

C++11/14 version:

```

1 template <class T> constexpr T absolute(T arg) {
2     return arg < 0 ? -arg : arg;
3 }
4
5 template <class T>
6 constexpr enable_if_t<is_floating_point<T>::value, boolreturn absolute(a - b) < static_cast<T>(0.000001);
9 }
10 template <class T>
11 constexpr enable_if_t<!is_floating_point<T>::value, boolreturn a == b;
14 }

```

As you see, there's a use of **enable_if**. It's very similar to our **str** function. The code tests if the type of input numbers is **is_floating_point**. Then, the compiler can remove one function from the overload resolution set.

And now, let's look at the C++17 version:

```

1 template <class T> constexpr T absolute(T arg) {
2     return arg < 0 ? -arg : arg;

```

```

3   }
4
5   template <class T>
6   constexpr auto precision_threshold = T(0.000001);
7
8   template <class T> constexpr bool close_enough(T a, T b) {
9     if constexpr (is_floating_point_v<T>) // << !!
10      return absolute(a - b) < precision_threshold<T>;
11    else
12      return a == b;
13 }
```

Wow...so just one function that looks almost like a normal function.

With nearly “normal” if :)

`if constexpr` evaluates `constexpr` expression at compile time and then discards the code in one of the branches.

But it’s essential to observe that the discarded code has to have the correct syntax. The compiler will do the basic syntax scan, but then it will skip this part of the function in the template instantiation phase.

That’s why the following code generates a compiler error:

```

template <class T> constexpr bool close_enough(T a, T b) {
  if constexpr (is_floating_point_v<T>)
    return absolute(a - b) < precision_threshold<T>;
  else
    return aaaa == bxxxx; // compiler error - syntax!
}

close_enough(10.04f, 20.f);
```

Checkpoint: Can you see some other C++17 features that were used here?

You can play with the code @Compiler Explorer

47.3.1.1 Adding Concepts in C++20

But wait...it’s 2021, so why not add some concepts? :)

Up to C++20, we could consider template parameters to be something like a `void*` in a regular function. If you wanted to restrict such a parameter, you had to use various techniques discussed in this article. But with Concepts, we get a natural way to restrict those parameters.

Have a look:

```

template <typename T>
requires std::is_floating_point_v<T>
constexpr bool close_enough20(T a, T b) {
  return absolute(a - b) < precision_threshold<T>;
}
```

```
constexpr bool close_enough20(auto a, auto b) {
    return a == b;
}
```

As you can see, the C++20 version switched to two functions. Now, the code is much more readable than with `enable_if`. With concepts, we can easily write our requirements for the template parameters:

```
requires std::is_floating_point_v<T>
```

`is_floating_point_v` is a type-trait (available in `<type_traits>` library) and as you can see the `requires` clause evaluates boolean constant expressions.

The second function uses a new generalized function syntax, where we can omit the `template<>` section and write:

```
constexpr bool close_enough20(auto a, auto b) { }
```

Such syntax comes from generic lambdas. This is not a direct translation of our C++11/14 code as it corresponds to the following signature:

```
template <typename T, typename U>
constexpr bool close_enough20(T a, U b) { }
```

Additionally, C++20 offers a terse syntax for concepts thanks to constrained `auto`:

```
constexpr bool close_enough20(std::floating_point auto a,
                             std::floating_point auto b) {
    return absolute(a - b) <
precision_threshold<std::common_type_t<decltype(a), decltype(b)>>;
}

constexpr bool close_enough20(std::integral auto a, std::integral auto b) {
    return a == b;
}
```

Alternatively, we can also put the name of the concept instead of a typename and without the `requires` clause:

```
template <std::is_floating_point T>
constexpr bool close_enough20(T a, T b) {
    return absolute(a - b) < precision_threshold<T>;
}
```

In this case, we also switched from `is_floating_point_v` into a concept `floating_point` defined in the `<concepts>` header.

See the code here: [@Compiler Explorer](#)

Ok, how about another use case?

47.3.2 Use case 2 - computing the average

Let's stay in some “numeric” area, and now we'd like to write a function that takes a vector of numbers and returns an average.

Here's a basic use case:

```
std::vector ints { 1, 2, 3, 4, 5};
std::cout << Average(ints) << '\n';
```

Out function has to:

- Take floating-point numbers or integral types.
- It returns double.

In C++20, we can use ranges for such purposes, but let's treat this function as our playground and test case to learn.

Here's a possible version with Concepts:

```
template <typename T>
requires std::is_integral_v<T> || std::is_floating_point_v<T>
constexpr double Average(const std::vector<T>& vec) {
    const double sum = std::accumulate(vec.begin(), vec.end(), 0.0);
    return sum / static_cast<double>(vec.size());
}
```

For the implementation, we need to restrict the template parameter to be integral or floating-point.

We don't have a predefined concept that combines floating point and integral types, so we can try writing our own:

```
template <typename T>
concept numeric = std::is_integral_v<T> || std::is_floating_point_v<T>;
```

And use it:

```
template <typename T>
requires numeric<T>
constexpr double Average2(std::vector<T> const &vec) {
    const double sum = std::accumulate(vec.begin(), vec.end(), 0.0);
    return sum / static_cast<double>(vec.size());
}
```

Or we can also make it super short:

```
constexpr double Average3(std::vector<numeric auto> const &vec) {
    const double sum = std::accumulate(vec.begin(), vec.end(), 0.0);
    return sum / static_cast<double>(vec.size());
}
```

We can also rewrite it with C++14 `enable_if`

```
template <typename T>
std::enable_if_t<std::is_integral_v<T> || std::is_floating_point_v<T>, double>
Average4(std::vector<T> const &vec) {
    const double sum = std::accumulate(vec.begin(), vec.end(), 0.0);
    return sum / static_cast<double>(vec.size());
}
```

See the working code [@Compiler Explorer](#)

47.3.3 Use case 3 - a factory with variable arguments

In the item 18 of Effective Modern C++ Scott Meyers described a function called `makeInvestment`:

```
template<typename... Ts>
std::unique_ptr<Investment>
makeInvestment(Ts&&... params);
```

This is a factory method that creates derived classes of `Investment` and the main advantage is that it supports a variable number of arguments!

For example, here are the proposed types:

```
1 class Investment {
2 public:
3     virtual ~Investment() { }
4
5     virtual void calcRisk() = 0;
6 };
7
8 class Stock : public Investment {
9 public:
10     explicit Stock(const std::string&) { }
11
12     void calcRisk() override { }
13 };
14
15 class Bond : public Investment {
16 public:
17     explicit Bond(const std::string&, const std::string&, int) { }
18
19     void calcRisk() override { }
20 };
21
22 class RealEstate : public Investment {
```

```

23 public:
24     explicit RealEstate(const std::string&, double, int) { }
25
26     void calcRisk() override { }
27 };

```

The code from the book was too idealistic and didn't work - it worked until all your classes have the same number and types of input parameters:

Scott Meyers: Modification History and Errata List for Effective Modern C++:

The **makeInvestment** interface is unrealistic because it implies that all derived object types can be created from the same types of arguments. This is especially apparent in the sample implementation code, where our arguments are perfect-forwarded to all derived class constructors.

For example if you had a constructor that needed two arguments and one constructor with three arguments, then the code might not compile:

```

// pseudo code:
Bond(int, int, int) { }
Stock(double, double) { }
make(args...)
{
    if (bond)
        new Bond(args...);
    else if (stock)
        new Stock(args...)
}

```

Now, if you write **make(bond, 1, 2, 3)** - then the else statement won't compile - as there no **Stock(1, 2, 3)** available! To work, we need something like static if that will work at compile-time and reject parts of the code that don't match a condition.

Some posts ago, with the help of one reader, we came up with a working solution (you can read more in Nice C++ Factory Implementation 2).

Here's the code that could work:

```

1 template <typename... Ts>
2 unique_ptr<Investment>
3 makeInvestment(const string &name, Ts&&... params)
4 {
5     unique_ptr<Investment> pInv;
6
7     if (name == "Stock")
8         pInv = constructArgs<Stock, Ts...>(forward<Ts>(params)...);
9     else if (name == "Bond")
10        pInv = constructArgs<Bond, Ts...>(forward<Ts>(params)...);

```

```

11     else if (name == "RealEstate")
12         pInv = constructArgs<RealEstate, Ts...>(forward<Ts>(params)...);
13
14     // call additional methods to init pInv...
15
16     return pInv;
17 }
```

As you can see, the “magic” happens inside `constructArgs` function.

The main idea is to return `unique_ptr<Type>` when Type is constructible from a given set of attributes and `nullptr` when it’s not.

47.3.3.1 Before C++17

In my previous solution (pre C++17) we used `std::enable_if` and it looked like that:

```

// before C++17

template <typename Concrete, typename... Ts>
enable_if_t<is_constructible<Concrete, Ts...>::value, unique_ptr<Concrete>>
constructArgsOld(Ts&&... params)
{
    return std::make_unique<Concrete>(forward<Ts>(params)...);
}

template <typename Concrete, typename... Ts>
enable_if_t<!is_constructible<Concrete, Ts...>::value, unique_ptr<Concrete> >
constructArgsOld(...)
{
    return nullptr;
}
```

`std::is_constructible` see cppreference.com - allows us to quickly test if a list of arguments could be used to create a given type.

In C++17 there’s a helper:

```
is_constructible_v = is_constructible<T, Args...>::value;
```

So we could make the code shorter a bit…

Still, using `enable_if` looks ugly and complicated. How about a C++17 version?

47.3.3.2 With `if constexpr`

Here’s the updated version:

```

template <typename Concrete, typename... Ts>
unique_ptr<Concrete> constructArgs(Ts&&... params)
{
```

```

if constexpr (is_constructible_v<Concrete, Ts...>)
    return make_unique<Concrete>(forward<Ts>(params)...);
else
    return nullptr;
}

```

Super short!

We can even extend it with a little logging features, using fold expression:

```

1 template <typename Concrete, typename... Ts>
2 std::unique_ptr<Concrete> constructArgs(Ts&&... params)
3 {
4     cout << __func__ << ":" ;
5     // fold expression:
6     ((cout << params << ", "), ...);
7     cout << "\n";
8
9     if constexpr (std::is_constructible_v<Concrete, Ts...>)
10        return make_unique<Concrete>(forward<Ts>(params)...);
11    else
12        return nullptr;
13 }

```

Cool…right? :)

All the complicated syntax of `enable_if` went away; we don't even need a function overload for the `else` case. We can now wrap expressive code in just one function.

`if constexpr` evaluates the condition, and only one block will be compiled. In our case, if a type is constructible from a given set of attributes, then we'll compile the `make_unique` call. If not, then `nullptr` is returned (and `make_unique` is not even instantiated).

47.3.3.3 C++20

With concepts we can easily replace `enable_if`:

```

1 // C++20:
2 template <typename Concrete, typename... Ts>
3 requires std::is_constructible_v<Concrete, Ts...>
4 std::unique_ptr<Concrete> constructArgs20(Ts&&... params) {
5     return std::make_unique<Concrete>(std::forward<Ts>(params)...);
6 }
7
8 template <typename Concrete, typename... Ts>
9 std::unique_ptr<Concrete> constructArgs20(...) {
10    return nullptr;
11 }

```

But I wonder if that's better? I think in this case, `if constexpr` looks much better and easier to follow.
Here's the working code @Compiler Explorer

47.3.4 Use case 4 - real-life projects

`if constexpr` is not only cool for experimental demos, but it found its place in production code.

If you look at the open-source implementation of STL from the MSVC team, we can find several instances where `if constexpr` helped.

See this Changelog: <https://github.com/microsoft/STL/wiki/Changelog>

Here are some improvements:

- Used `if constexpr` instead of tag dispatch in: `get<I>()` and `get<T>()` for pair. #2756,
- Used `if constexpr` instead of tag dispatch, overloads, or specializations in algorithms like `is_permutation()`, `sample()`, `rethrow_if_nested()`, and `default_searcher`. #2219, `<map>` and `<set>`'s common machinery. #2287 and few others,
- Used `if constexpr` instead of tag dispatch in: Optimizations in `find()`. #2380, `basic_string(first, last)`. #2480
- Improved vector's implementation, also using `if constexpr` to simplify code. #1771

Let's have a look at improvements for `std::pair`:

Untag dispatch `get` for pair by frederick-vs-ja • Pull Request #2756 • microsoft/STL

Before C++17 benefits, the code looked as follows:

```

1 template <class _Ret, class _Pair>
2 constexpr _Ret _Pair_get(_Pair& _Pr, integral_constant<size_t, 0>) noexcept {
3     // get reference to element 0 in pair _Pr
4     return _Pr.first;
5 }
6
7 template <class _Ret, class _Pair>
8 constexpr _Ret _Pair_get(_Pair& _Pr, integral_constant<size_t, 1>) noexcept {
9     // get reference to element 1 in pair _Pr
10    return _Pr.second;
11 }
12
13 template <size_t _Idx, class _Ty1, class _Ty2>
14 _NODISCARD constexpr tuple_element_t<_Idx, pair<_Ty1, _Ty2>>&
15     get(pair<_Ty1, _Ty2>& _Pr) noexcept {
16     // get reference to element at _Idx in pair _Pr
17     using _Rtype = tuple_element_t<_Idx, pair<_Ty1, _Ty2>>;
18     return _Pair_get<_Rtype>(_Pr, integral_constant<size_t, _Idx>{});
19 }
```

And after the change, we have:

```

1 template <size_t _Idx, class _Ty1, class _Ty2>
2 _NODISCARD constexpr tuple_element_t<_Idx, pair<_Ty1, _Ty2>>&
3 get(pair<_Ty1, _Ty2>& _Pr) noexcept {
4     // get reference to element at _Idx in pair _Pr
5     if constexpr (_Idx == 0) {
6         return _Pr.first;
7     } else {
8         return _Pr.second;
9     }
10 }
```

It's only a single function and much easier to read! No need for tag dispatch with the `integral_constant` helper.

In the other library, this time related to SIMD types and computations (popular implementation by Agner Fog), you can find lots of instances for `if constexpr`:

<https://github.com/vectorclass/version2/blob/master/instrset.h>

One example is the mask function:

```

1 // zero_mask: return a compact bit mask mask for zeroing using AVX512 mask.
2 // Parameter a is a reference to a constexpr int array of permutation indexes
3 template <int N>
4 constexpr auto zero_mask(int const (&a)[N]) {
5     uint64_t mask = 0;
6     int i = 0;
7
8     for (i = 0; i < N; i++) {
9         if (a[i] >= 0) mask |= uint64_t(1) << i;
10    }
11    if constexpr (N <= 8) return uint8_t(mask);
12    else if constexpr (N <= 16) return uint16_t(mask);
13    else if constexpr (N <= 32) return uint32_t(mask);
14    else return mask;
15 }
```

Without `if constexpr` the code would be much longer and potentially duplicated.

47.4 Wrap up

Compile-time `if` is an amazing feature that significantly simplifies templated code. What's more, it's much more expressive and nicer than previous solutions: tag dispatching or `enable_if` (SFINAE). Now, you can easily express your intentions similarly to the “run-time” code.

We also revised this code and examples to work with C++20! As you can see, thanks to concepts, the code is even more readable, and you can “naturally” express requirements for your types. You also gain a few syntax shortcuts and several ways to communicate such restrictions.

In this article, we've touched only basic expressions, and as always, I encourage you to play more with this new feature and explore.

47.4.1 Going back…

And going back to our str example:

Can you now rewrite the str function (from the start of this article) using `if constexpr`? :) Try and have a look at my simple solution @CE.

47.4.2 Even more

You can find more examples and use cases for `if constexpr` in my C++17 Book: C++17 in Detail @Leanpub or @Amazon in Print

Chapter 48

The memory subsystem from the viewpoint of software: how memory subsystem affects software performance

• Ivica Bogosavljević  2022-7-26 / 2022-8-17  ★★★

48.1 Part 1

We at *Johnny's Software Lab LLC* are experts in performance. If performance is in any way concern in your software project, feel free to contact us.

We talked about the memory subsystem in one way or another many times (e.g. Instruction-level parallelism in practice: speeding up memory-bound programs with low ILP or Memory consumption, dataset size and performance: how does it all relate?). And not without reason: memory performance is a limiting factor in many types of computation.

In this post we are going to experiment with memory subsystem from the software viewpoint. We will write small programs (kernels) that illustrate different aspects of the memory subsystem to explore how parts of the memory subsystem behave under various loads.

The idea of the post is to give you an overview of how different types of software interact with the memory subsystem and how you can take advantage of this knowledge. We intentionally omit many details of the memory subsystems, because they often differ between vendors or architectures. Instead, we focus on effects that are common to all architectures and most visible from the software viewpoint. This is the first post in the series of three posts. In the second post, we will talk about cache conflicts, cache line size, hiding memory latency, TLB cache, vectorization and branch prediction. And in the third post, we will talk about multithreading and the memory subsystem.

48.1.1 A short introduction to the memory hierarchy

CPU's are fast and memory is slow. That's why CPU's have several levels of cache memories: *level 1, level 2 and level 3 (also called last-level) caches* to speed up access to commonly used data. Level 1 data cache memories are the fastest, but the smallest, followed by L2 and L3. Here is the snapshot of the memory

hierarchy in my laptop taken by Hardware Locality tool lstopo:

Machine (15GB total)

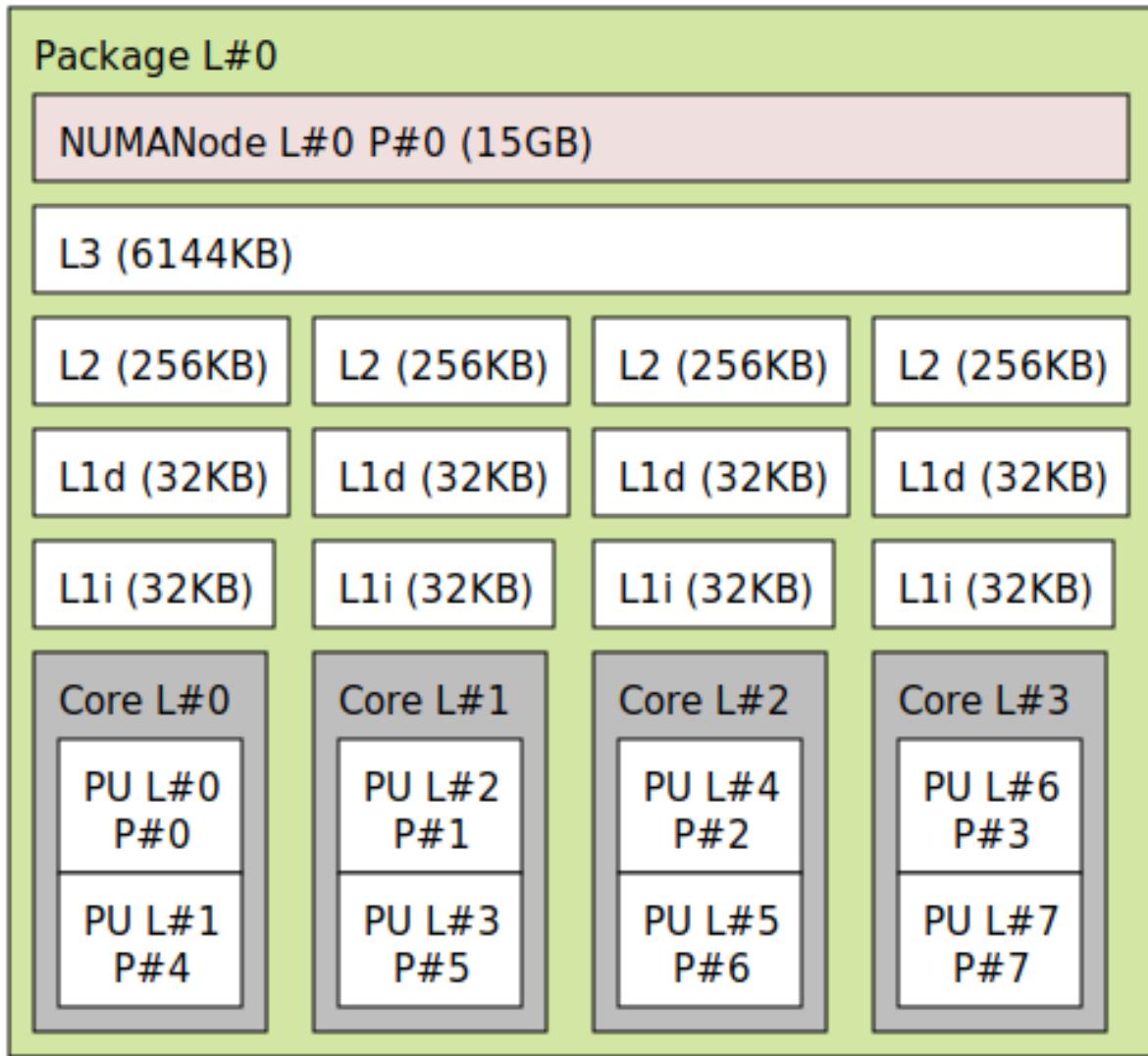


图 48.1: Hardware Topology Tool for my CPU

The system displayed has 32 kB of per-core L1 cache, 256 kB of per-core L2 cache and 6 MB of L3 cache common to all cores.

A piece of data that has been recently accessed will be in the cache. After some time of non use, it will get *evicted*, i.e. moved to the slower parts of the memory subsystem. For this reason, a repeated access to a small hash map (e.g. 8 kB) will be faster than a repeated access to a large hash map (e.g. 8 MB). This is because 8 kB of data completely fits the L1 cache, whereas 8 MB of data only partially fits the slowest L3 cache.

An important part of the memory hierarchy is the *data prefetcher*: a circuitry in the CPU that prefetches data before it's needed. This works however, only if the memory access pattern is predictable. This makes iterating through a vector much faster than iterating through a linked list.

Another important aspect of the memory subsystem is the cache line. A cache line is typically 64 bytes

in size, and data is brought from the memory to the data caches in blocks of 64 bytes. What this means is that if your code touches a certain byte, with high probability the few next bytes or few previous bytes will be already in the cache and access to them will be fast.

And lastly, there is TLB cache. Modern systems don't use physical addresses, but only virtual addresses, and there is a small TLB cache used to speed up the translation from physical to virtual addresses. A typical memory page size is 4 kB and there is one translation entry for each page. If the program is accessing many different pages in a short period of time, the translation entry will be missing in the TLB cache and will need to be fetched from the memory. This is a very expensive memory operation, and it actually does happen with random memory accesses on very large data sets - imagine accessing a hash map which is 256 MB in size.

Like what you are reading? Follow us on LinkedIn, Twitter or Mastodon and get notified as soon as new content becomes available. Need help with software performance? Contact us!

48.1.2 The experiments

Here are few experiments that demonstrate the effects of the memory subsystem on software performance. For these experiments we use three test systems: Raspberry Pi 4 (embedded), Intel's CORE i5 (desktop) and Intel's Xeon Gold (server). We use three different systems for two reasons. The first reason is to illustrate the difference in memory subsystem for each of them and second, to illustrate the complex interplay of CPU and memory and their effect on performance.

Here are the specifications for the systems

- *Desktop system:* Intel Core i5-10210U CPU, one socket in one NUMA domain, 4 cores, 2 threads per core. Each core has 32kB of L1i cache, 32kB of L1d cache and 256kB of L2 cache. There is a total of 6MB of L3 cache shared by all the cores.
- *Embedded system:* Broadcom BCM2711, ARM Cortex-A72, one socket in one NUMA domain, 4 cores, 1 thread per core. Each core has 16kB L1i cache and 16kB L1d cache. There is a total of 1MB of L2 cache shared by all the cores.
- *Server system:* Intel Xeon Gold 5218 CPU, two sockets in two NUMA domains, 16 cores per socket, 2 threads per core. Each core has 32kB of L1i cache, 32kB of L1d cache and 1MB of L2 cache. There is a total of 22MB of L3 cache common to all the cores in the socket.

48.1.2.1 Performance and levels of cache - sequential access

In the first experiment, we have two small kernels, one doing additions and one doing divisions. They all iterate sequentially through arrays. Here are they:

```

1 void sum(double* a, double* b, double* out, int n) {
2     for (int i = 0; i < n; i++) {
3         out[i] = a[i] + b[i];
4     }
5 }
6
7 void div(double* a, double* b, double* out, int n) {

```

```

8     for (int i = 0; i < n; i++) {
9         out[i] = a[i] / b[i];
10    }
11 }
```

The kernels are very simple. Each of them performs two loads, one arithmetic operation and one store. We repeat the experiment for various array sizes and repeat counts. In total, we have 256 M iterations, which means that if our array has 1M elements, we will repeat the experiment 256 times, but if it has 128 M, we will repeat the experiments only two times.

Here are the runtimes depending on the array size for the desktop system. The total number of accessed elements is always the same, 256M.

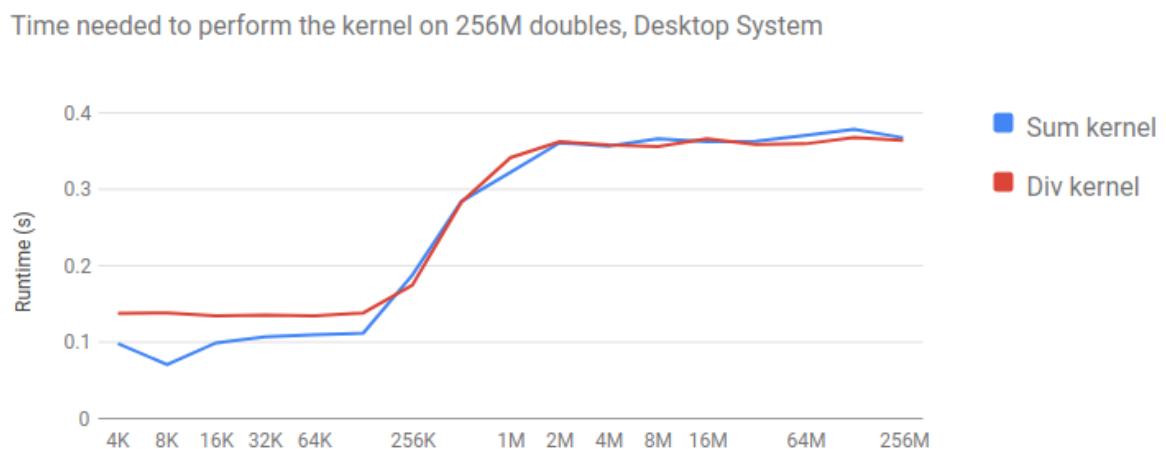


图 48.2: Array size

In modern CPUs, additions is relatively cheap compared to divisions. And this is the case while the data is coming either from L1, L2 or L3 cache. When the dataset doesn't fit any of the caches and needs to be fetched from the memory, both kernels run at about the same speed. The kernels run about 3 to 3.5 times slower when the data is fetched from the memory compared to when it is fetched from the data caches.

For smaller arrays, the bottleneck is the CPU and therefore the SUM kernel is faster. For larger arrays, the bottleneck is the memory and both kernels run at about the same speed.

Here are the same graphs for the server system:

And the same graph for the embedded system:

As far as the server system, the graph looks similarly to the desktop system. The SUM kernel is faster for the small dataset, but they even out as soon as the dataset doesn't fit the L1 cache. As far as the embedded system, the graph looks a bit different. The SUM kernel exhibit similar properties as with the other two systems, being faster when data is served from the cache and slowing down once data starts being served from the main memory. But the runtime of DIV kernel is mostly independent from the dataset size, and this is because division is very slow on this chip and therefore the bottleneck is always the CPU regardless of the dataset size.

Time needed to perform the kernel on 256M doubles, Server System

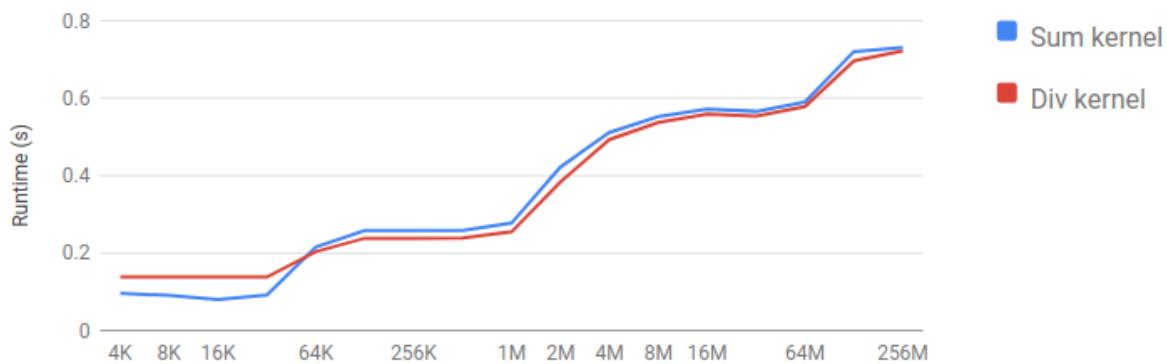


图 48.3: Array size

Time needed to perform the kernel on 256M doubles, Embedded System

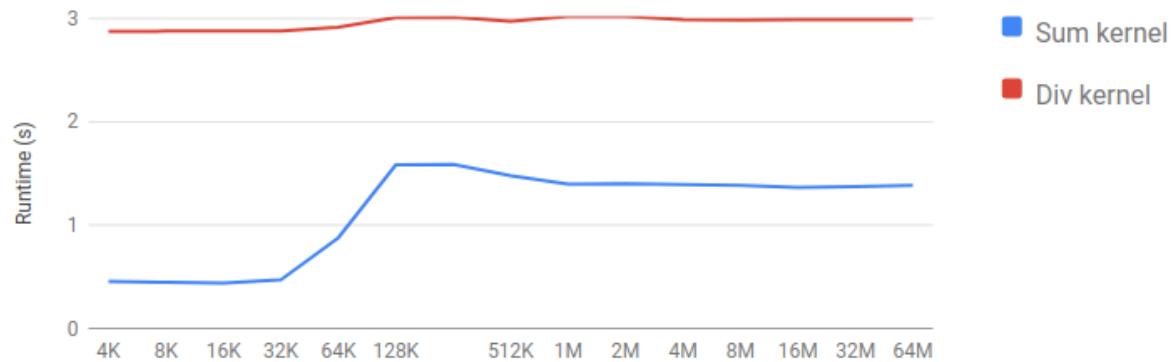


图 48.4: Array size

48.1.2.2 Memory access pattern, levels of cache and performance

In the previous experiment, we were concerned only with sequential memory accesses. Sequential memory accesses are very common in image processing, video processing, matrix manipulation, etc. Other than sequential memory accesses, we have two important types of memory accesses:

- *Strided memory accesses*: whereas sequential memory access is iterating neighboring memory cells from left to right (or right to left), strided memory accesses skip a fixed number of elements. The number of elements that is skipped is called stride. For example, we can iterate an array but touch every odd element of the array. In this case the stride is two. Another example is iterating an array of classes. If the size of class is X, and we are touching a data member of X.y, from the hardware viewpoint we are iterating the data members of X.y with the stride sizeof(X).
- *Random memory accesses*: random memory access happen either when we dereference a pointer (through * or -> operators), or when we are accessing elements of an array, but the index is not directly related to the iterator variable (e.g. a[index[i]] is a random access to a, and index[i] is a sequential access to i if iterator increments using i++).

These are the main differences between sequential, strided and random memory accesses:

- Sequential memory accesses are the most efficient, because they completely use all the data from the data cache lines. In addition, the data prefetcher can prefetch the data because the access pattern is predictable.
- Strided memory accesses come next in efficiency. The utilization of cache line is smaller, and the bigger the stride, the smaller the utilization. But the access pattern is predictable, so the code still profits from data prefetcher.
- Random memory accesses are the least efficient, because neither prefetching is possible and the cache utilization rate is typically low.

Having covered the memory access patterns, let's move on to experimenting. We want to measure how dataset size, memory access pattern, data caches and runtime are connected. To do this, we use the following kernel:

```
int calculate_sum(int* data, int* index, int len) {
    int sum = 0;
    for (int i = 0; i < len; i++) {
        sum += data[indexes[i]];
    }
    return sum;
}
```

By just observing the code, the access to data using `data[index[i]]` is random. But for the purpose of this experiment, we fill out the index array to obtain sequential, strided or random accesses.

We measure runtimes, but also we measure memory throughput (in MB/s) and total memory volume (in MB). Here are first the runtimes:

Time needed to perform the kernel on 64M ints

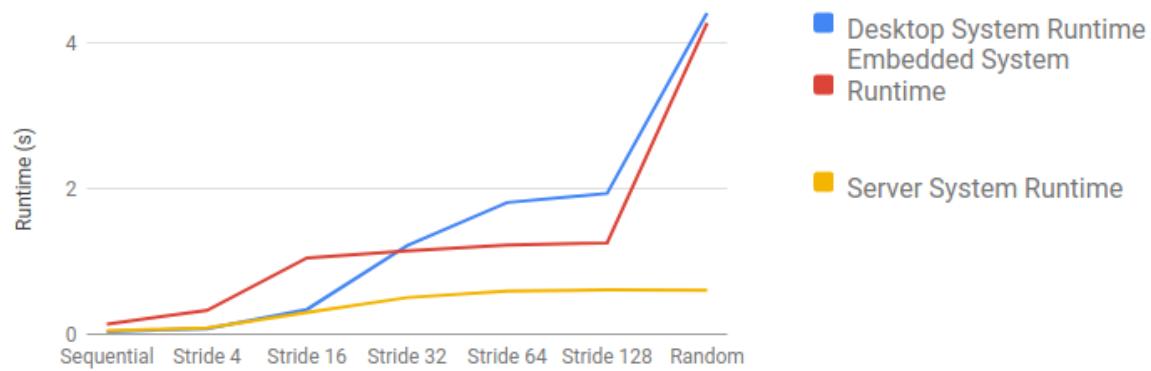


图 48.5: Memeory Access Type

A few interesting observations:

- Performance is very much related to the access pattern, the best being sequential, then small strides, then large strides and followed by random memory access pattern

- With a sequential memory access pattern, all three systems behave similarly. The desktop system is a bit faster than the server system. The Embedded system is the slowest, but this is to be expected due to the smaller and cheaper CPU.
- As the access pattern worsens, so does the performance.
- With random access, both the desktop system and the embedded systems are equally slow. The reason is that the bottleneck is no longer the CPU, but exclusively the memory
- The server system is much more resistant to worsening of access pattern. This is probably because the memory subsystem being much faster and having a higher throughput than the others.

Let's measure memory throughput and data volume. Unfortunately, the data is only available for the desktop system, but this is enough to help us illustrate some facts. First memory throughput:

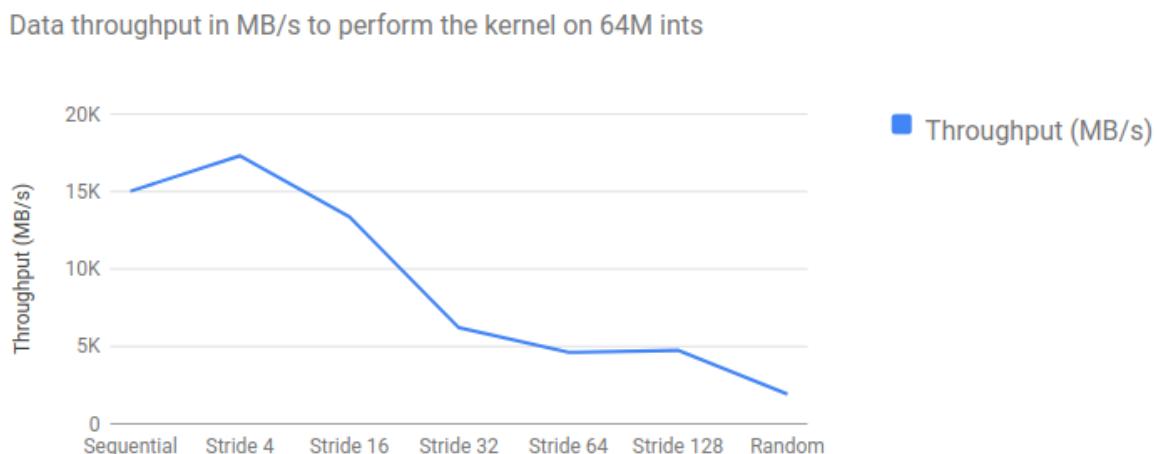


图 48.6: Memory Access Type

General trends is that the memory throughput is maximum for sequential accesses and small stride accesses, and then begins to fall quickly. Notice that the maximum throughput is not for sequential accesses, but small strided accesses. With sequential accesses the kernel is still not completely memory bound and there is some unused throughput left.

We know that our program uses exactly 512MB of data. Since there are two arrays of integers, each with 64M elements, this sums it up to exactly 512MB of memory. From the viewpoint of the CPU, the total memory data volume transferred will depend a lot on the memory access pattern, because of the cache line organization.

With the sequential access, the amount of data is as expected (0.54GB). But as the stride grows, so does the data volume. The difference between the minimum and maximum data volume is 17x. The main reason is the cache line nature of memory transfers. Data is brought from the memory to the data caches in blocks of 64 bytes¹. Let's calculate cache line utilization: number of bytes consumed vs number of bytes fetched. Here are the numbers:

¹Typical cache line sizes are 64 and 128, but other sizes are possible in specialized systems (32 with embedded systems or 256 with high-performance systems).

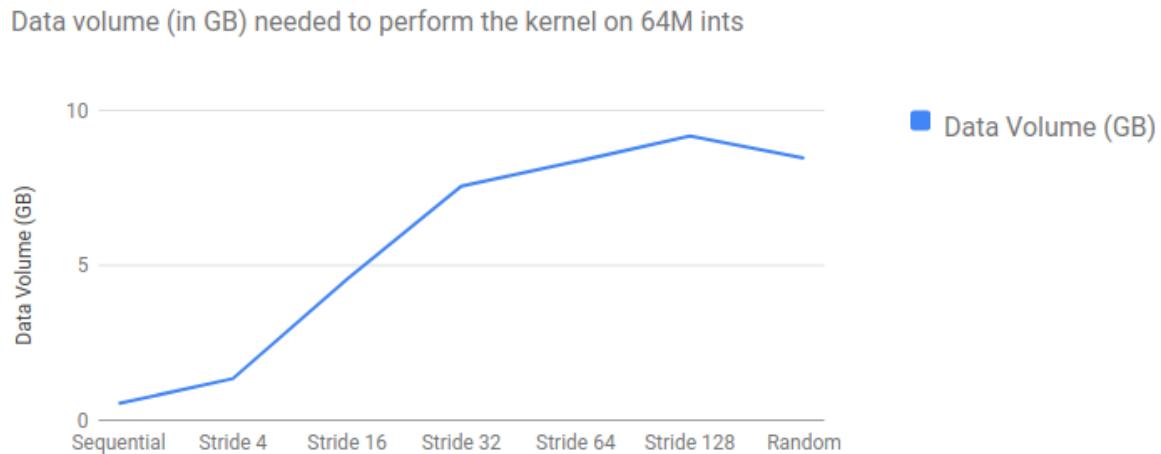


图 48.7: Memeory Access Type

Cache Line Utilization for an Array of Integers	
Access Type	Cache Line Utilization
Sequential	100% (64/64)
Stride 2	50% (32/64)
Stride 4	25% (16/64)
Stride 8	12.5% (8/64)
Stride 16	6.25% (4/64)
Stride 32	6.25% (4/64)

For larger strides, many cache lines will have to be brought from the memory to the CPU caches several times. This increases data volume. If only cache line utilization were responsible for the increase in data volume, then we could expect a maximum volume of $256\text{MB} + 256\text{MB} \times 16 = 4352\text{ MB}$. We see higher data loads here, so implies that additional data volume comes from other places as well².

48.1.2.3 Loads vs stores performance

What is more expensive, loading data from the memory or storing it back? There are two rationales: the first one is that loads are more expensive, because until the data is loaded, no operation depending on the loaded value can continue (see the post about instruction level parallelism). The second one is that stores are more expensive, because stores in principle require two memory transfers (loading the cache line, modifying only some part of it and then storing the modified cache line).

To measure this, we use two small kernels that load from the memory and store to it. We measure the performance of loading and storing for both sequential memory accesses and strided memory accesses with stride 2. Here is the source code of the memory load (read) kernel:

```
template <int stride = 1>
double test_read(double* in, int n) {
    double sum = 0.0;
    for (int i = 0; i < n; i+=stride) {
```

² It is very difficult to say where the additional data volume comes from, because of the complexity of the memory subsystem and the inaccuracy of the measurement system.

```

        sum += in[i];
    }
    return sum;
}

```

This kernel reads n elements from the array in. We store the results in variable sum, because we don't want the compiler to optimize away the reads.

The second kernel is the store (write) kernel. Here is the source code:

```

template <int stride = 1>
void test_write(double* in, int n, double v) {
    for (int i = 0; i < n; i+=stride) {
        in[i] = v;
    }
}

```

This code is equivalent of memset function, which writes the same value to all elements of an array.

To test the memory subsystem, we repeat the test on arrays between 8K elements and 256K elements. For each array size, we repeat the experiment so that in total we get 1G memory accesses for the sequential kernels and 512M accesses for the stride 2 kernels.

We are interested in the runtime ratio between the corresponding load and store kernels. If the value is above 1, this means the writes are faster. If the value is below 1, this means that the reads are faster. Here are the numbers for all three systems:

Read vs write runtime ration for various kernel sizes

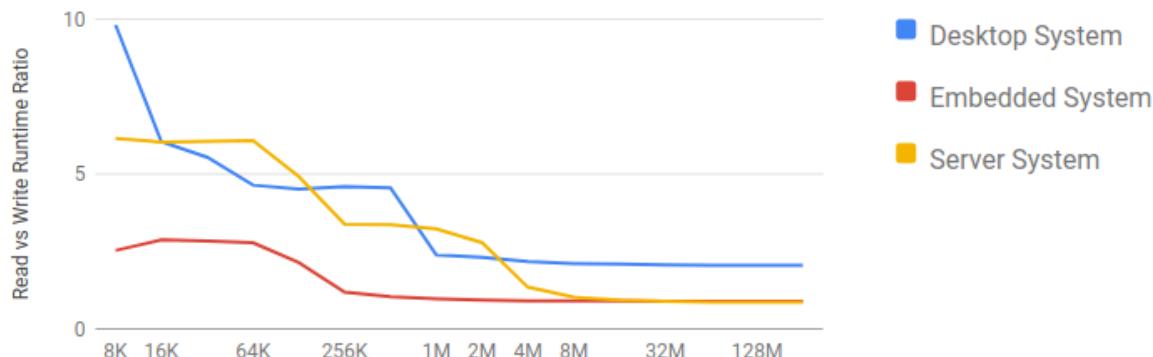


图 48.8: Array size

When the array is small (i.e. the data is mostly served from the caches), stores are faster by a large margin. This is to be expected because in real conditions loading stalls the CPU until the data is fetched. Memory stores are never instruction dependencies so the CPU has the ability to delay them. Loading is faster when the size of the array becomes larger than 8M elements, on embedded and server systems, but not on the desktop system.

The absolute results are very different between the three platforms, but the general trend is similar: stores are faster when going through the cache, but once the workload doesn't fit the data caches, they are roughly the same speed.

Here is the same graph, but this time when the kernels are running with stride 2.

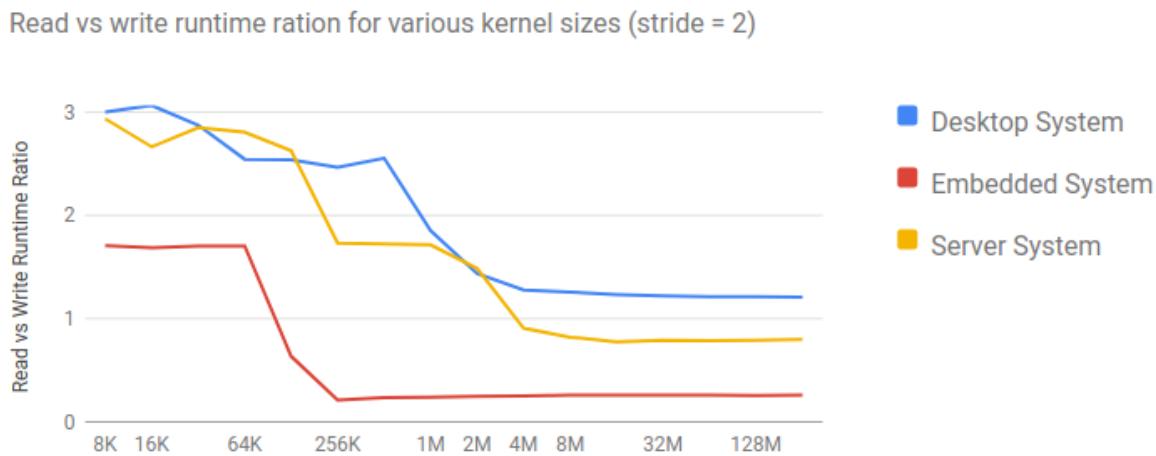


图 48.9: Array size

We see different results here. Again, stores are faster than loads, up to a certain dataset size. When the dataset doesn't fit the data caches, stores become slower than loads. This is especially true for the embedded system.

Why is this happening? As already said, loads are almost always parts of the instruction dependency chains, because the loaded data is used later. In contrast, stores are never part of the dependency chains. So stores should be in principle faster.

However, the store consists of reading the cache line from the memory to the data cache, modifying it, and then writing the modified cache line back to the memory. So, when a piece of data is not in the data cache, storing data is more expensive than loading data. But, this only works for non-sequential accesses (strided and others).

With sequential accesses, the CPU doesn't need to fetch the whole line from the memory. If the whole cache line is modified, several stores can be merged into one large memory write that modifies the whole cache line in the process of write combining. This allows skipping one memory transaction and makes stores faster.

When the stores are not served from the main memory, then stores are almost always faster than corresponding loads. But once they start getting served from the main memory, they can be marginally faster, or slower, depending on the architecture. They are more slower when with strided accesses than with sequential accesses (2.05 vs 1.21 for the desktop system, 0.86 vs 0.8 for the server system and 0.86 vs 0.26 for the embedded system).

The bottom line is that loading is more expensive than storing, so having a good memory access pattern for loading is more important than having a good memory access pattern for storing³. But the details really depend on the architecture and are likely to change with newer architectures.

48.1.2.4 Arithmetic Intensity

Another interesting effect of the interplay of the memory subsystem and the CPU is arithmetic intensity and performance. By arithmetic intensity, we mean the ratio between the number of bytes loaded/stored from

³e.g. with loop tiling, where you can have only good memory access pattern only for either loads and stores, but not for both

the memory and the number of arithmetic operations in the single iteration of the loop. Here is the example kernel we are going to use for the testing:

```
template<int p>
void pow_of_2_array(double* in, double* out, int n) {
    for (int i = 0; i < n; i++) {
        out[i] = pow_of_2<p>(in[i]);
    }
}
```

Notice the call to the function `pow_of_2<p>(...)`. This function is written using C++ templates. The number of multiplications performed by the function is p . To test, we use the values for p : 1, 4, 7, ..., 25, 28.

Since the number of loads and stores doesn't change, increasing p also means increasing the arithmetic intensity of the loop. We want to see how the runtime changes by increasing the number of multiplications.

We are also interested in dataset size, so we vary the array size from 4K elements up to 256M elements. In total, the kernel performs 1G iterations, corresponding to 16 GB data load, 8 GB data stores and from 1 until 28 multiplications, depending on the value of p .

Here are the runtimes for the desktop system:

Runtime depending on arithmetic intensity and array size, desktop system

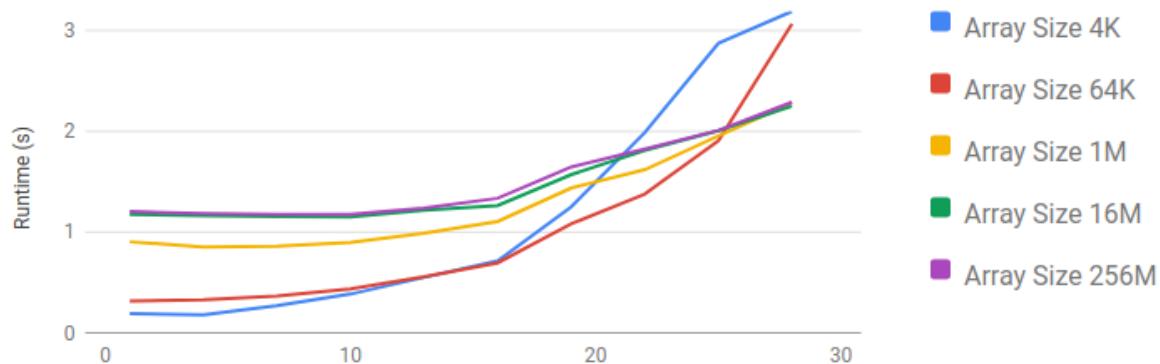


图 48.10: Multiplication Count

As expected, when the number of multiplication is low, the runtime doesn't depend on the multiplication count. This is because loading and storing data is the bottleneck. However, at some point (e.g. 4 multiplications for array size 64K, 10 multiplications for array size 1M and 13 multiplications for array size 256M) the runtime starts to increase. The increase in runtime corresponds to a point where the loop stops being memory bound and starts being computationally bound. The smaller the dataset, the smaller the number of multiplication that is needed for the loop to become computationally bound.

We observe a similar pattern in the server system:

And the similar pattern in the embedded system:

With the embedded system, due to the limitation in the chip, the loop becomes very quickly computationally bound, because the price of multiplication is much higher than the price of memory data loads and stores.

Runtime depending on arithmetic intensity and array size, server system

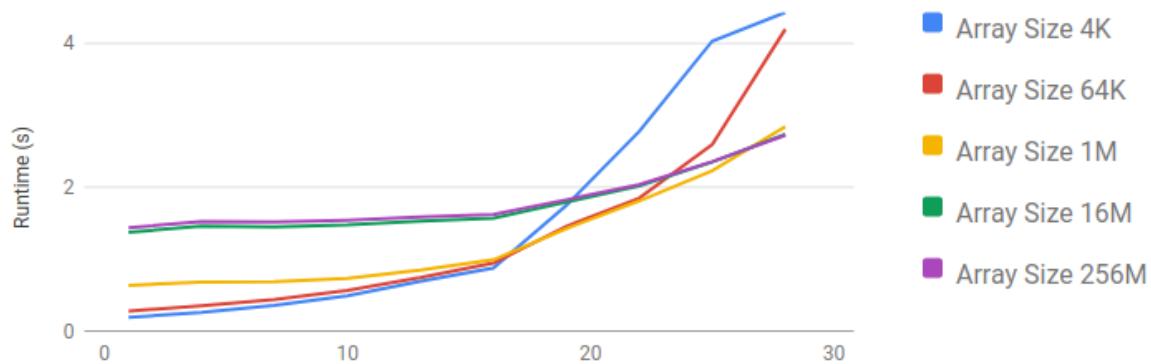


图 48.11: Multiplication Count

Runtime depending on arithmetic intensity and array size, embedded system

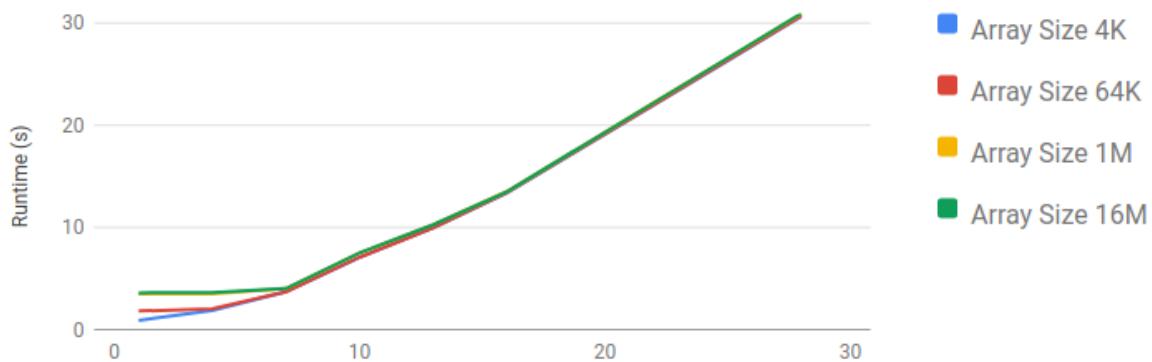


图 48.12: Multiplication Count

The relationship between arithmetical intensity and performance has been formalized in the *roofline model*. The roofline model tells you the memory and computational limit of the target architecture, and also tells you if your hot loop is hitting the computational or memory limits. More about the roofline model can be found [here](#), [here](#) and [here](#).

48.1.3 Final Words

Although the exact details of architectures differ, most of our observations apply to all three investigated architectures:

- Working with smaller datasets is faster than working with larger datasets.
- A predictable memory access pattern is faster than an unpredictable memory access pattern.
- The sequential memory access pattern is faster than the strided access pattern. The bigger the stride, the bigger the difference.
- Memory stores are typically faster than memory loads, but this can depend on the dataset size and the underlying architecture.

- More arithmetically intense code puts less pressure on the memory subsystem. Between loading and storing data, if the code does a lot of computations, the bottleneck will not be the memory subsystem but the CPU.

This is the first post in the series where we experiment with the memory subsystem. In the second post, we will talk about cache conflicts, cache line size, hiding memory latency, TLB cache, vectorization and branch prediction. And in the third post, we will talk about multithreading and the memory subsystem. Stay tuned!

48.2 Part 2

We at [Johnny's Software Lab LLC](#) are experts in performance. If performance is in any way concern in your software project, feel free to contact us.

In the previous post we started a series of experiments to see how the memory subsystem affects software performance. In a certain sense, that post is a prerequisite for this post. There we experimented with performance and different levels of cache, memory access pattern and performance, load vs store performance as well as arithmetic intensity. In this post, we experiment with:

48.2.1 The Experiments

48.2.1.1 Cache Line

As mentioned in the previous post, data is fetched from the memory in chunks corresponding to cache line size. Typically this is 64 bytes, but can be smaller (embedded systems) or larger (high-performance systems). If the size of chunk is N, then chunks are aligned at N bytes.

When a chunk of memory is brought up from the memory to the caches, access to any byte inside the chunk will generally be fast. Access to a byte outside the chunk will require another memory transfer.

To test the effect of cache line size we a struct called `test_struct` and two small kernels:

```

1 template <int CacheLineSize = CACHE_LINE_SIZE>
2 struct test_struct {
3     unsigned char first_byte;
4     unsigned char padding[CACHE_LINE_SIZE - 2];
5     unsigned char last_byte;
6 };
7
8 void initialize_struct(test_struct<CACHE_LINE_SIZE> * p, int n) {
9     for (int i = 0; i < n; i++) {
10        p[i].first_byte = i;
11        p[i].last_byte = n - i;
12    }
13 }
14
15 int sum_struct(test_struct<CACHE_LINE_SIZE> * p, int n, std::vector<int>& index_array) {
16    int sum = 0;

```

```

17     for (int i = 0; i < n; i++) {
18         int index = index_array[i];
19         sum += p[index].first_byte + p[index].last_byte;
20     }
21
22     return sum;
23 }
```

The `test_struct` struct is peculiar. Its size is the size of the cache line, which is 64 in all our architectures. There are two important members that we are using `first_byte` and `last_byte`. Between them, there is padding to make the struct exactly 64 bytes in size. If the struct is aligned at 64 bytes, then the `first_byte` and the `last_byte` will be in the same cache line. For all other possible alignments, `first_byte` and `last_byte` will be in different cache lines.

The values of `test_struct` are stored in a vector (array). There are two functions we test. The first one is called `initialize_struct` which goes through the array of structs *sequentially* and initializes all the fields. The second is called `sum_struct` and performs *random accesses* in the array of structs using the vector `index`. We will see that the memory access pattern (sequential vs random) is very important for performance.

For the testing we use two arrays. The first one is aligned at 64 bytes. The second one is misaligned from 64 bytes by one byte. Again, we test for various array sizes while keeping the number of iterations constant: 256M iterations. Here are the runtimes for the first function `initialize_struct` that traverses the array sequentially:

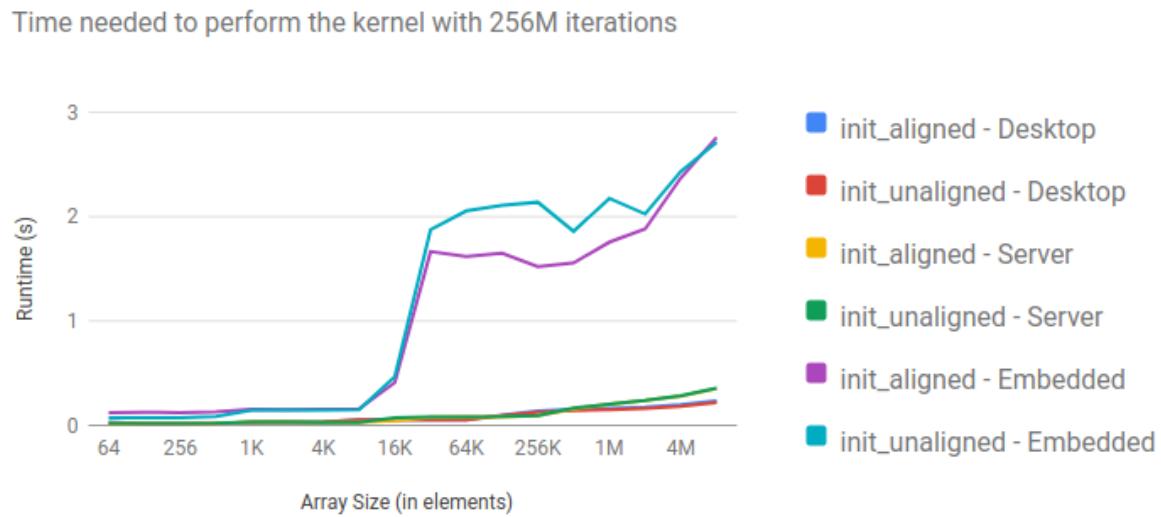


图 48.13: Array Size(in elements)

The server and the desktop system don't care if the data is aligned or unaligned; the runtimes are the same. For the embedded system, there is a critical array size where the runtimes differ. This is between 32K (2MB) and 2M (128MB).

Here are the runtimes for the second function `sum_struct` which traverses the array randomly:

In all three architectures, we observe the same behavior. Once the dataset doesn't fit the data caches, the aligned version becomes faster.

The question: why is the impact higher with random accesses compared to sequential accesses? With

Time needed to perform the kernel with 256M iterations

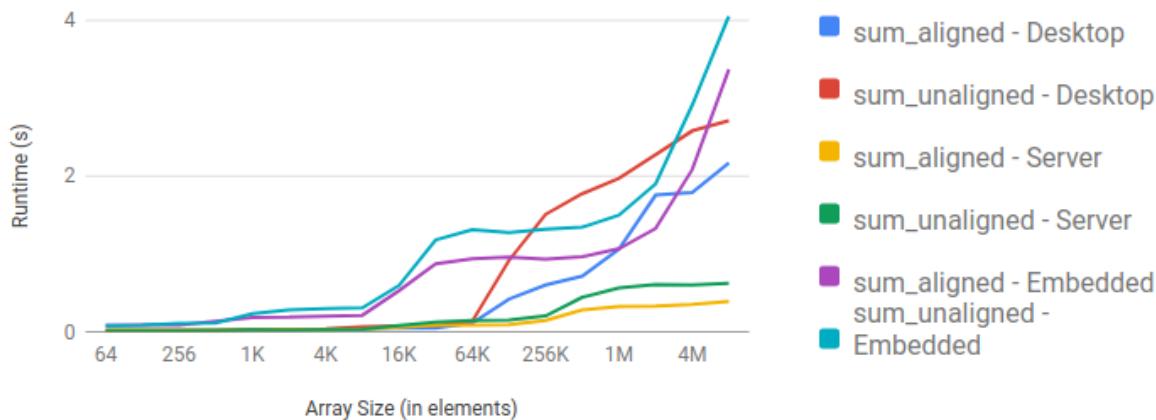


图 48.14: Array Size(in elements)

sequential accesses, even when the piece of data is split between the two cache lines, there is some cache line reuse because, even though `p[i].first_byte` and `p[i].last_byte` don't share the same cache line, `p[i].last_byte` and `p[i+1].first_byte` share the same cache line. This is not the case for random memory accesses.

Unaligned accesses also increase the amount of data that the system needs to transfer from the memory to the CPU. Of course, this only applies to random accesses and `sum_struct` function. For `initialize_struct`, the amount of data that is transferred is independent if the array is aligned or not.

Here is the data volume for the desktop and the embedded system:

Total memory data volume needed to perform the kernel with 256M iterations

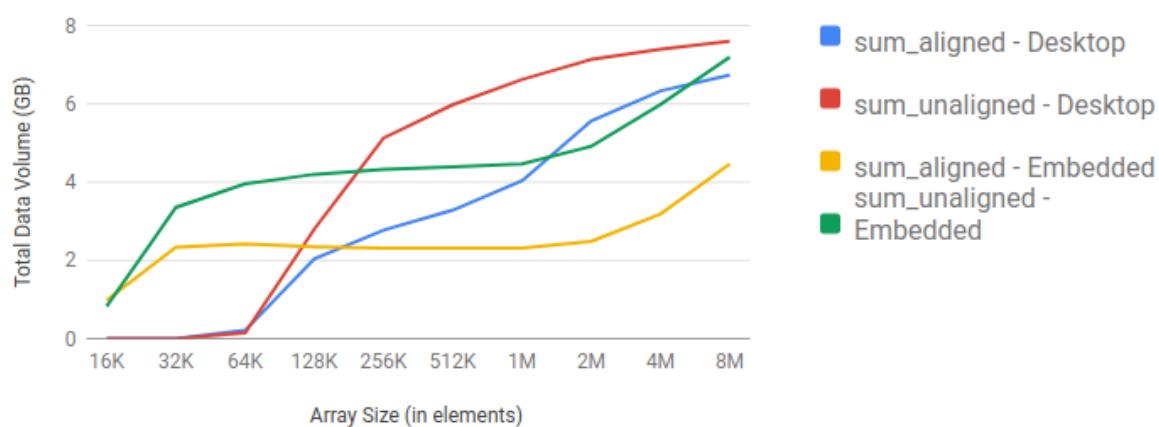


图 48.15: Array Size(in elements)

Once the dataset doesn't fit the data caches, the total data volume is almost two times higher for the unaligned version compared to the aligned version.

Cache line size needs to be taken into account when designing efficient random memory access data structures, such as trees, hash maps or linked list. Ideally, the size of a single element of the data structure should be a multiple of cache line size and also properly aligned. This will ensure that a single element is

never split among two cache lines, which improves speed and decreases the total data volume exchanged between the memory and the CPU.

48.2.2 Hiding Memory Latency

The RAM memory, as well as each individual cache in the memory hierarchy, has two important properties:

- *Memory throughput*, which measures the amount of data that the CPU can get from the memory/LLC/L2/L1 cache per unit of time. Typically, it is measured in MB/s or GB/s.
- *Memory latency*, which measures the time that passes between the CPU issuing a memory access request, and the memory/LLC/L1/L1 cache responding with the data. Typically, it is measured in nanoseconds or cycles.

The story is not this simple, however. Because of its out-of-order nature, the CPU can keep several pending memory accesses (loads or stores). And the memory can process several interleaved load requests. The CPU can therefore issue several load/store requests in parallel. The effect of this is “hiding” memory latency. The latency might be long, but since the CPU is issuing many requests in parallel, the effect of memory latency will be hidden.

To explain it more visually, let’s use an analogy with a transport company. Imagine you have a transport company with only one truck. You want to transport your goods to a city which is 3 hours away. The latency is therefore three hours. You can only transport one truckload every six hours. Now imagine your company has 6 trucks. You can dispatch a new truck every half an hour. Although the latency is the same, having 6 trucks effectively hides it.

Back to our story. With regards to memory latency, there is a software limitation about how much latency the CPU can hide. If the piece of code is essentially *following a chain of pointers*, that is, it is dereferencing the current pointer to get the address of the next pointer, there are very few loads/stores that the CPU can do in parallel. It needs to wait for the current load to finish in order to issue the next store. We say this code has *a low instruction level parallelism*, and we dealt this in the this post.

Following a chain of pointers is not that uncommon. E.g. traversing a linked list, a binary tree or a hash map with collisions in the bucket results in following a chain of pointers. One of the ways to increase the available instruction level parallelism is by *interleaving additional work, e.g. by following two chains of pointers or following one chain of pointers but doing more work for each pointer in the chain*.

To illustrate this effect, we take a kernel that traverses several linked lists at once. The sizes of linked lists vary between 1K elements (16kB) and 64M elements (1GB), and the number of traversed linked lists vary between 1 and 11. Here is the kernels source code:

```

1 double calculate_min_sum(std::vector<elem_t*> heads) {
2     std::vector<elem_t*> currents = heads;
3     std::vector<double> sums(heads.size(), 0.0);
4     int list_cnt = heads.size();
5
6     while (currents[0] != nullptr) {
7         for (int i = 0; i < list_cnt; i++) {

```

```

8         sums[i] += i * currents[i]->value;
9         currents[i] = currents[i]->next;
10    }
11   }
12
13   double min = sums[0];
14   for (int i = 0; i < list_cnt; i++) {
15     min = std::min(sums[i], min);
16   }
17
18   return min;
19 }
```

We use the time needed to traverse a single linked list as a baseline. So, if there is no instruction level parallelism available, the relative time needed to traverse a single linked list 1 for one linked list, 2 for two linked lists, etc. However, with out-of-order execution, we will see a smaller runtime ratio.

Here are the relative times for the desktop system depending on the number of linked list traversed:

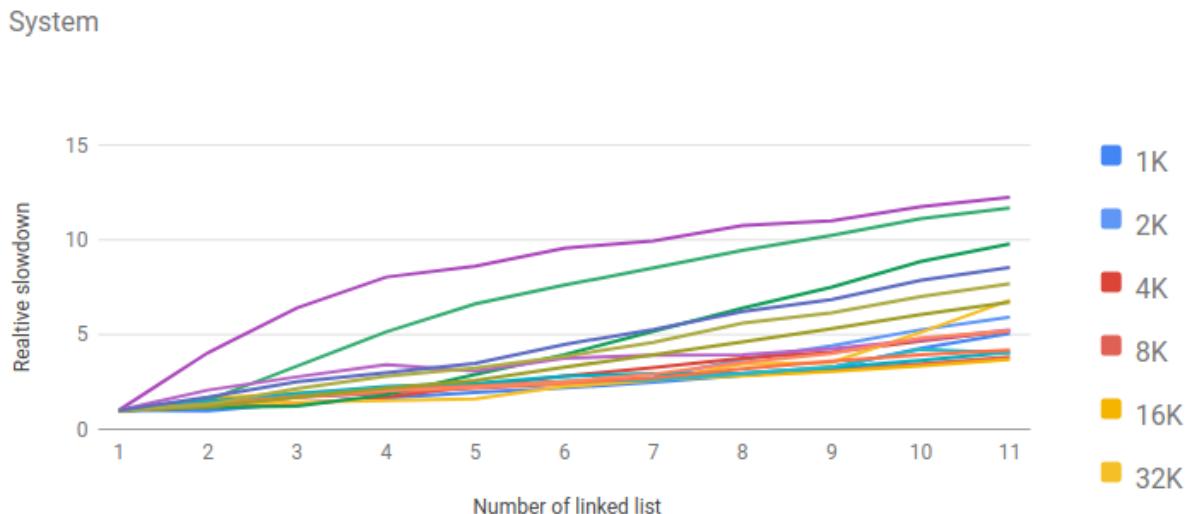


图 48.16: Number of linked list

For the most linked list sizes, the slowdown is not linear to the number of linked lists. The reason is as already explained: memory latency hiding. For the traversing 11 linked lists in parallel, the smallest slowdown is 3.64 (list size 16K or 256kB) and the largest slowdown is 12.2 (list size 256K or 4MB). The size of L3 cache is 6MB, so one linked list completely fits the L3 cache, but two linked lists don't. This is probably the explanation of the largest slowdown; the misses in the L3 cache are the most costly.

Here is the graph for the server system:

A similar effect is observed with the server system. The server system is also good in latency hiding. In fact, in comparison to the desktop system where the largest slowdown was 12.2 for 11 parallel linked lists, here the largest slowdown is 4.9 for the linked list with 512K elements (8MB in size). This CPU has 22 MB of cache, so the largest slowdown corresponds to the same case of exhausting the L3 cache.

Lastly, here is the same diagram for the embedded system:

System

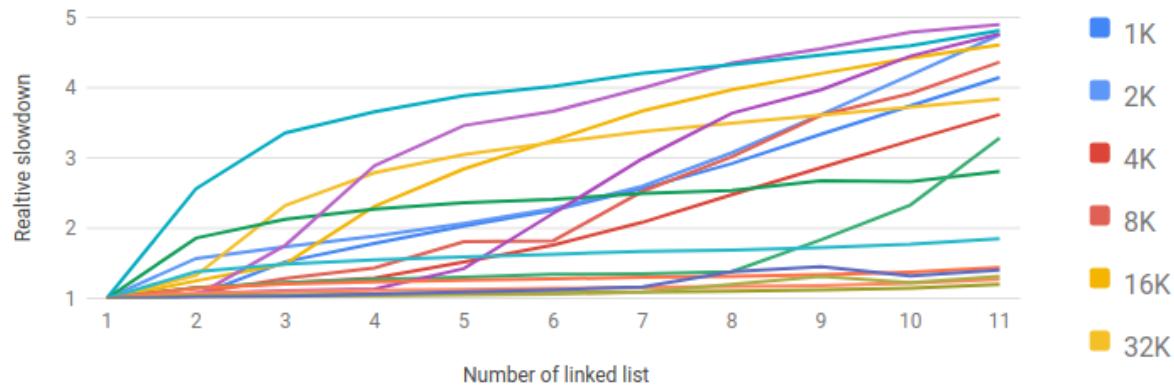


图 48.17: Number of linked list

Embedded System

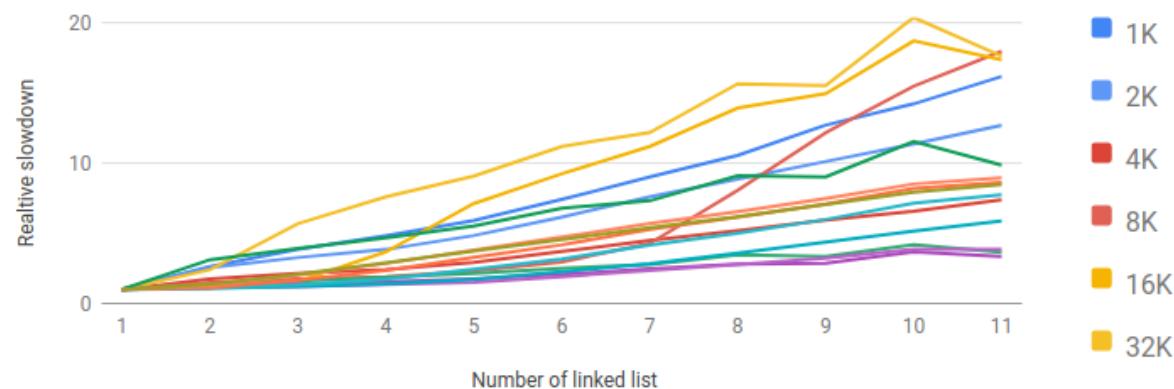


图 48.18: Number of linked list

The graph, however, paints a different picture of the embedded system. The worst slowdown is 20.4 for the kernel with 10 linked lists and 32K or 512kB. This system, although with an out-of-order CPU is in no way as good in hiding memory latency compared to the other two systems. It lies to both the CPU and the memory subsystem: the CPU can hold less pending loads/stores, and the memory subsystem has only limited parallelism.

In all the experiments we had until now, this one shows the largest differences between the three systems. Whereas the server system has a lot of hardware helping it hide memory latency, the situation is worse with the desktop system and much worse with the embedded systems⁴.

⁴ An important note: When each new linked list the data set size increases. Larger datasets don't fit the data caches well, which results in performance loss. If the dataset size doesn't increase with the additional work, then the results will be better.

48.2.3 TLB cache and large pages

TLB caches are small caches that speed up translation from virtual to physical addresses. There is one entry in the TLB cache for each memory page, which is most often 4kB. The size of this cache is quite limited, and therefore if the size of the dataset is larger than the size of this cache, this will result in performance penalty.

To give an illustration, on the desktop system under test, the size of this cache is 64 entries for L1 TLB (4-way associative) and 1536 entries for L2 TLB (6-way associative). So, the L1 cache can hold translation information for a block of memory 256kB in size and the L2 cache can hold the translation information for a block of memory 6MB in size. This is quite small.

If the code is doing random memory accesses in the block of data that is more than 256kB in size, it will experience L1 TLB cache misses. If the block of data is more than 6MB in size, the program will experience L2 TLB cache misses.

Problems with TLB cache misses really depend on the memory access pattern and data set size. Sequential access pattern and small strided access pattern rarely experience problems with TLB cache misses. On the other hand, random memory accesses on large datasets are very prone to TLB cache misses and performance losses.

Most modern CPUs support large memory pages, also known as huge memory pages. Instead of a memory page which is 4kB in size, there are larger pages, e.g. 2MB in size or 1GB in size. Using larger pages decreases the pressure on the TLB caches. To illustrate this, we use the same kernel from the previous post with memory access pattern section:

```

1 int calculate_sum(int* data, int* index, int len) {
2     int sum = 0;
3     for (int i = 0; i < len; i++) {
4         sum += data[indexes[i]];
5     }
6     return sum;
7 }
```

The values in the indexes array are created in such a way to create sequential, strided or random memory accesses. For the testing we use an array of 64M ints. We run the kernel with standard pages (4kB) and large pages (2 MB). Here are the results for the desktop system:

The graph looks very interesting. The difference in speed is very obvious for the random memory access (speed up of 2.8x), and doesn't exist at all for other access types.

Unfortunately, we couldn't run the experiment on other two systems, because large pages require a special system configuration, but we don't expect any difference in results with this kernel.

48.2.4 Cache Conflicts

Cache conflicts happen when two data blocks map to the same line in cache. A data block has an exact cache line where it can be stored. This is determined by higher N bits of the memory address accessing the data block. If that cache line is already occupied, it needs to be evicted from the data cache to the memory.

Cache conflicts typically appear when the data memory access pattern is a power of two: 256, 512, 1024, etc. It often happens when processing matrices whose size is a power of two, but it can also happen

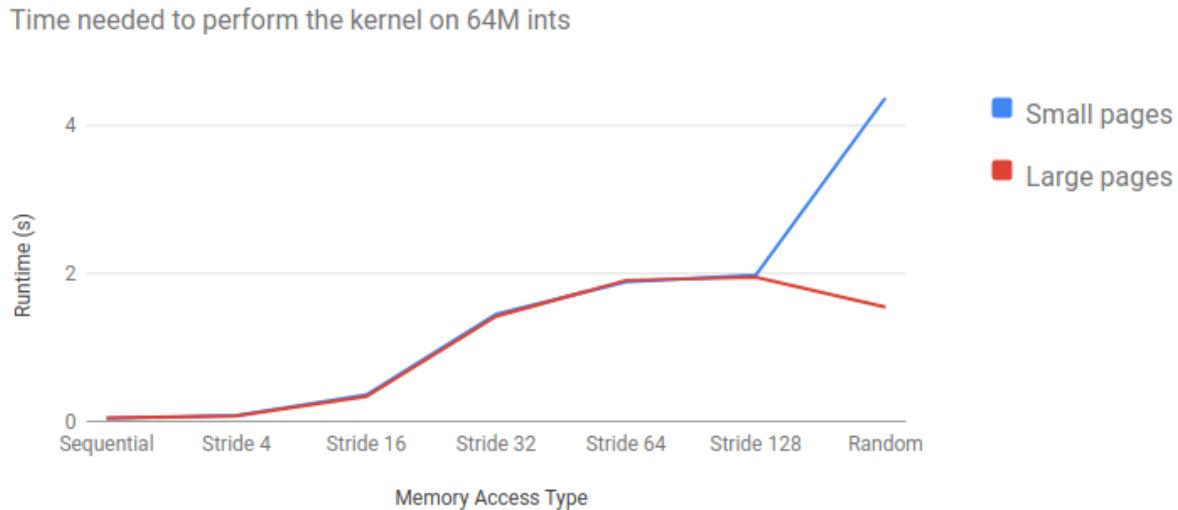


图 48.19: Memeory Accesses Type

when processing classes with the size of power or two. It can happen when processing two or more arrays, when the difference between the arrays start addresses is a power of two.

To illustrate the performance penalty of cache conflicts, we use matrix multiplication example:

```

1  for (int i = 0; i < n; i++) {
2      for (int j = 0; j < n; j++) {
3          c[i][j] = 0;
4          for (int k = 0; k < n; k++) {
5              c[i][j] = c[i][j] + a[i][k] * b[k][j];
6          }
7      }
8  }
```

The algorithm is quite simple, it's a triply nested loop that performs sequential access over the array `a[i][k]` and strided access over the array `b[k][j]`. The algorithmic complexity of this loop nest is $O(n^3)$, so we can expect that the runtime gets worse as the number `n` grows.

Here are the runtimes for the desktop system, for the value of `n` between 768 and 1280.

Although the general trend is upwards, as one might expect, we see a large spike in runtime for the case when the matrix size is 1024. This is the case that corresponds to a situation with a large amount of cache conflicts.

Here is the same graph for the server system:

There is a large spike for the matrix size 768 (3×256), 1024 (4×256) and 1280 (5×256), but there are also smaller spikes for matrix sizes 896 (7×128) and 1152 (9×128). So, as you can see, the code can have various amounts of cache conflicts which will incur smaller or larger performance penalty.

Here is the same graph for the embedded system:

The diagram is similar to previous diagrams and doesn't require further explanations.

The bad thing about cache conflicts is that, to my knowledge, there is no simple readily-available tool that detects cache conflicts.

Time needed to perform the matrix multiplication - Desktop System

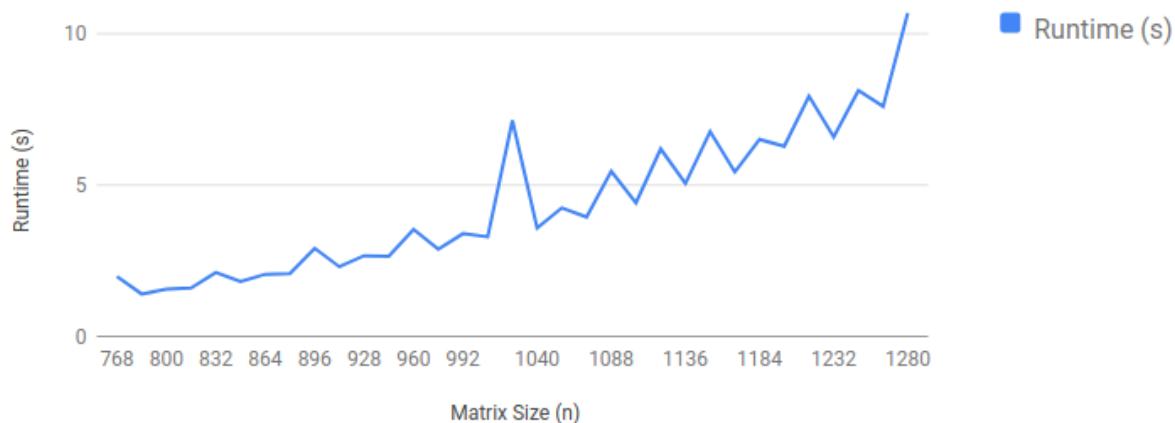


图 48.20: Matrix Size(n)

Time needed to perform the matrix multiplication - Server System



图 48.21: Matrix Size(n)

48.2.5 Vectorization

When the CPU is running vectorized code, instead of performing one operation per instruction, the CPU performs N (where N is typically 2, 4, 8, 16 or 32) operations per instruction. Vectorization is used to speed computations; ideally, the speedup should be N. With vectorization, the limiting factor in speedup is the memory subsystem. If the memory subsystem cannot deliver data fast enough, then vectorization will be slower than one would expect.

To measure the effect of memory subsystem on vectorization and software performance, we devise the following experiment. We use the two kernel from the previous post to measure the vectorization speedups. We compare the scalar vs vectorized version of the kernel. Here are the two kernels:

```

1 void sum(double* a, double* b, double* out, int n) {
2     for (int i = 0; i < n; i++) {
3         out[i] = a[i] + b[i];

```

Time needed to perform the matrix multiplication - Embedded System

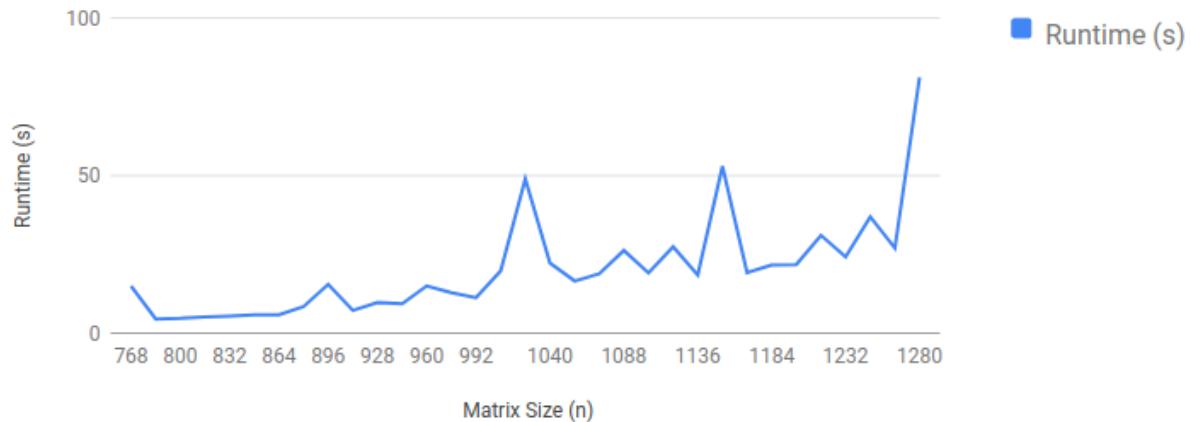


图 48.22: Matrix Size(n)

```

4      }
5  }
6  void div(double* a, double* b, double* out, int n) {
7      for (int i = 0; i < n; i++) {
8          out[i] = a[i] / b[i];
9      }
10 }

```

The kernels are very simple. Each of them performs two loads, one arithmetic operation and one store. We repeat the experiment for various array sizes and repeat counts. In total, we have 256 M iterations, which means that if our array has 1M elements, we will repeat the experiment 256 times, but if it has 128 M, we will repeat the experiments only two times.

Here is the runtime ratio between the scalar and vector version of the same kernel for the desktop system. In this case, the vectorization width is 4, which means that ideally the runtime ration should be 4:

Runtime ratio between the scalar and the vector system - Desktop System

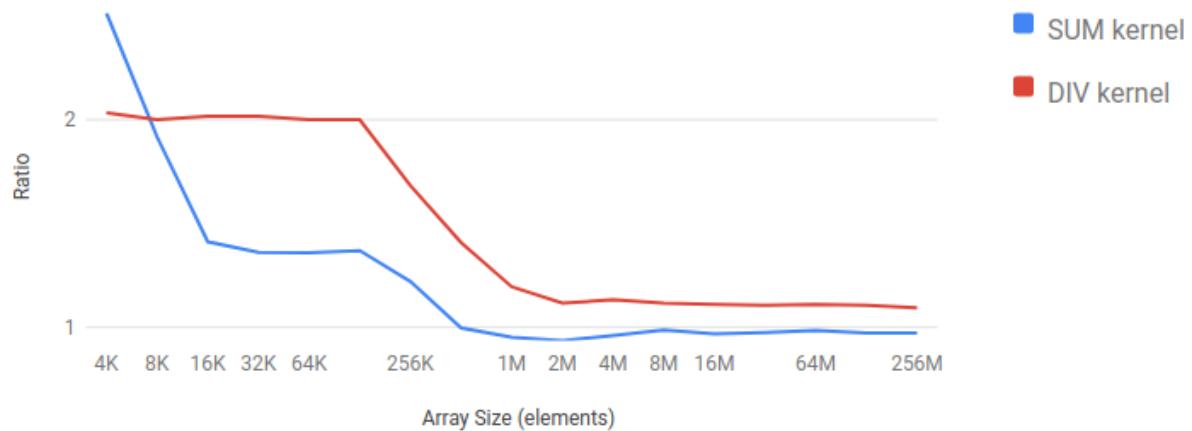


图 48.23: Array Size(elements)

Vectorization pays off until the data fits the data caches. Beyond that point, vectorization doesn't pay off at all for the SUM kernel, and pays off very little for the DIV kernel. This is because both loops have low arithmetic intensity⁵ so most of the time is spent waiting for data.

The same graph for the server system:

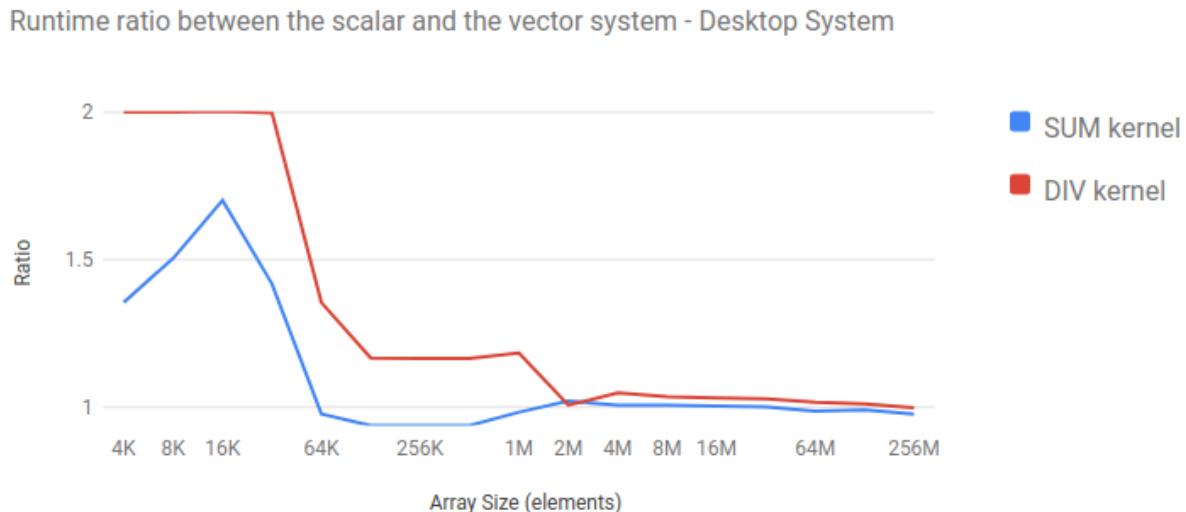


图 48.24: Array Size(elements)

This graph is very similar to the previous one and doesn't require additional explanation:

And finally, the same graph for the embedded system:

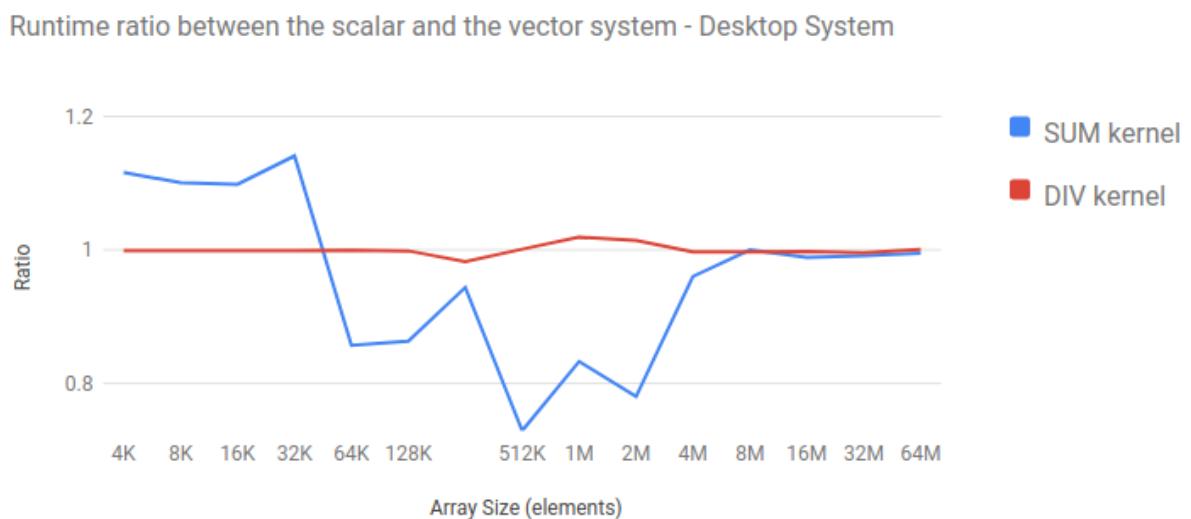


图 48.25: Array Size(elements)

Although this might seem like an error, I double checked the results and this is what reproduces consistently. SUM kernel has some vectorization benefits for really small array size. DIV vector doesn't care, vectorized or not, the results are always the same.

Bottom line, in the presence of limitations in the memory subsystem, vectorization is more or less useless. Embedded system shows no advantage for these small examples. Other systems show speed im-

⁵Arithmetic intensity is the ration of arithmetic operations to memory operations

provements with vectorization, but this speed improvement is limited and related to the data set size.

48.2.6 Branch Prediction

A branch predictor is a piece of CPU that predicts the outcome of the branch, before the condition is evaluated. This allows the CPU to speed up execution because it breaks the dependency chain; the CPU doesn't need to wait for the data, it can continue executing instructions. If the prediction is correct, then the CPU has saved some time because it has already done useful work. If not, the CPU needs to revert the instructions executed until that point and change the execution path. There is a certain performance penalty if the branch prediction is false, and this price depends on the CPU.

Sometimes, avoiding branches can lead to speed improvements. But, other times, it can lead to speed degradation. Why? If the CPU cannot evaluate the condition of a branch because the condition operands are not available, then the CPU can use branch prediction to move the work forward. In the presence of a high data cache miss rate, the price of not doing anything while waiting for the data is much higher than the price of doing something, and then reverting if it turns out to be useless.

In the cases of low data cache miss rates, branchless versions are typically faster. But in the cases of high data cache miss rates, branchfull versions are faster. To illustrate this, we use the example of binary search: we perform 256M searches on binary trees of various sizes. We use two algorithms, one is the branchless algorithm, and the other is branchfull algorithm.

Here is the runtime ratio between the branchfull and the branchless version for the desktop system:

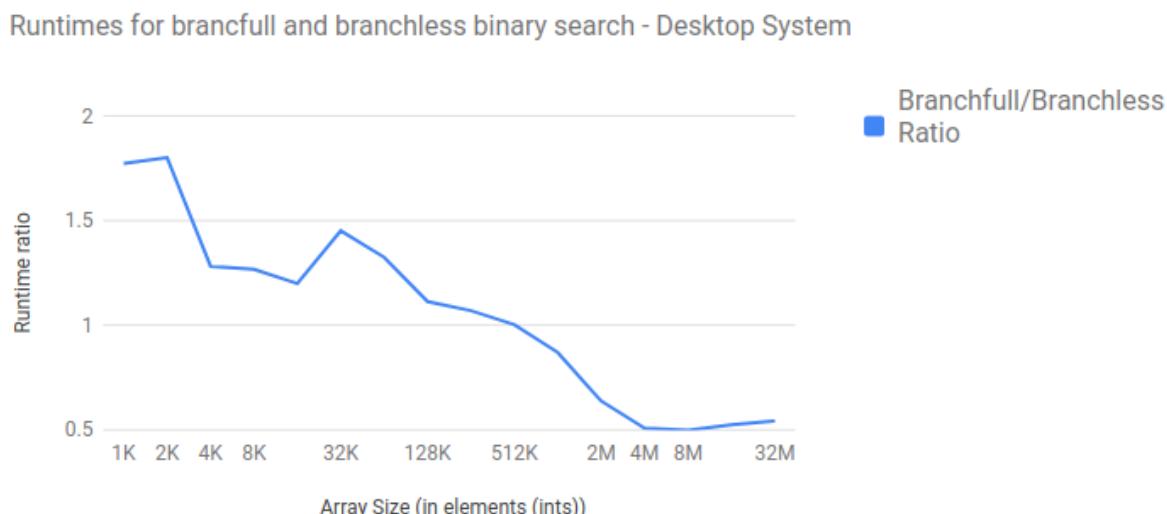


图 48.26: Array Size(in elements(ints))

Values greater than 1 mean that the branchless version is faster. Values smaller than 1 mean that the branchfull version is faster.

In the above graph, branchless version is faster while the binary search array fits the L1 or L2 data caches. Once it doesn't, the branchfull version becomes faster for the reasons already explained.

Here is the same graph for the server system:

A similar observation can be made for the server system.

The same graph for the embedded system:

The embedded system paints a different picture. While the data mostly fit the L1 cache, branchless

Runtimes for branchfull and branchless binary search - Server System

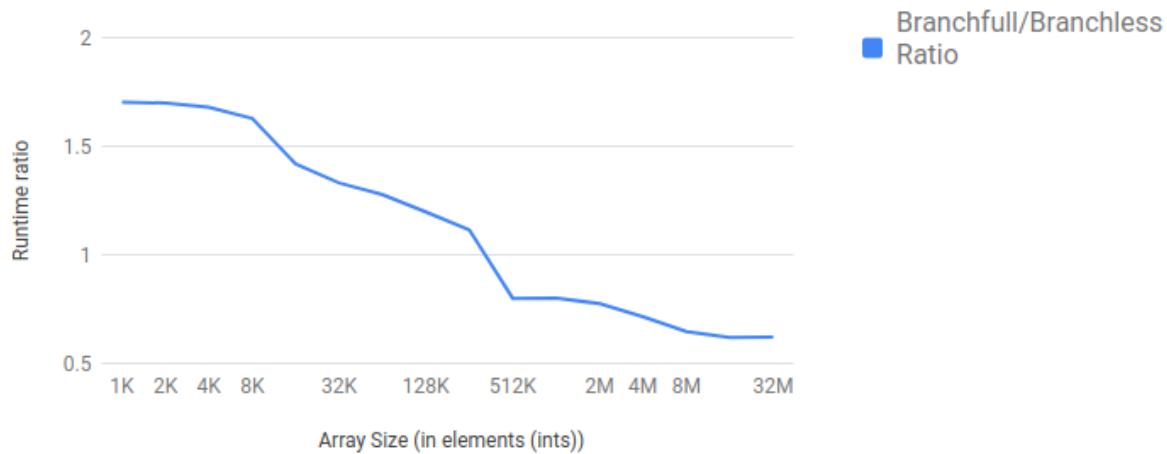


图 48.27: Array Size(in elements(ints))

Runtimes for branchfull and branchless binary search - Embedded System

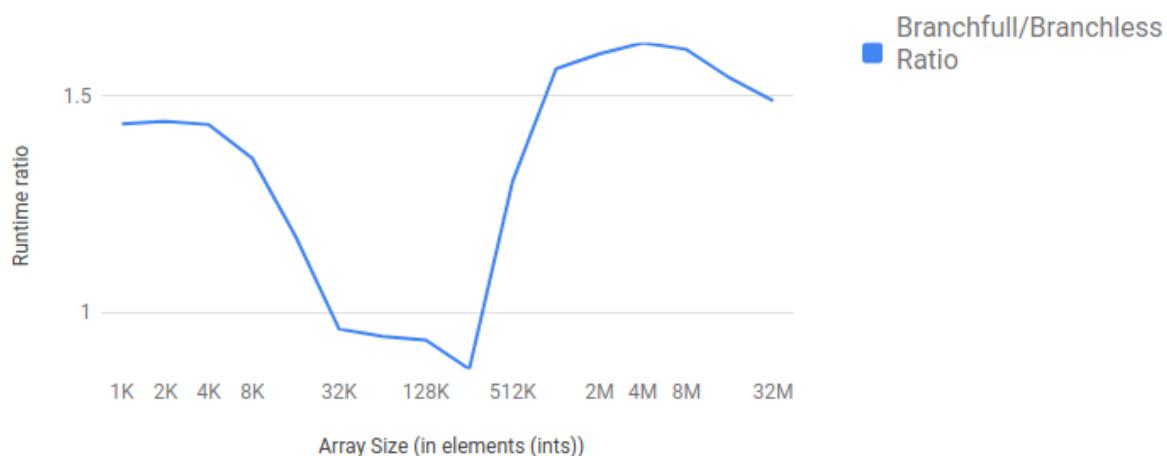


图 48.28: Array Size(in elements(ints))

version is faster. When the data doesn't fit the L1 cache, but fits the L2 cache, the branchfull version becomes faster. But quickly, as the data doesn't fit the data caches anymore, the branchless version becomes faster again.

I don't have an exact reason why this happens, but if I would have to guess, I would think that in the cases where the dataset doesn't fit the data caches anymore, there is a large stress on the RAM memory. Branch prediction wastes more memory because some loads will not be used. In contrast, branchless version only brings in the data that is actually useful. Decreased pressure on the RAM memory results in higher speeds for the embedded system.

Let's measure the total memory data volume for the embedded system⁶ for branchless and branchfull binary search. Here is the graph:

Branchless version brings in roughly two times less data from the memory to the CPU compared to the branchfull version. We can safely assume the same behavior can be observed in other two systems.

⁶We didn't have the corresponding counters to measure total memory data volume for the other two systems.

Total data volume for branchfull and branchless binary search - Embedded System

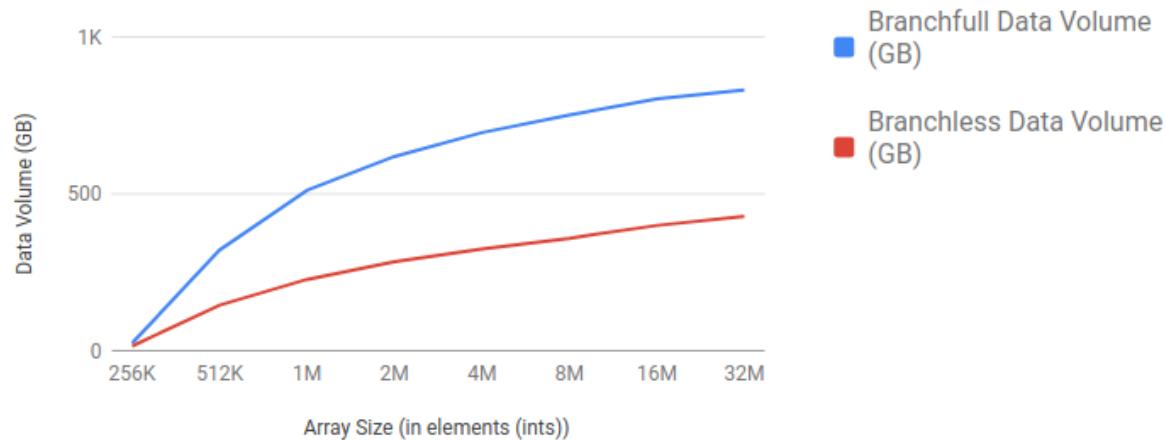


图 48.29: Array Size(in elements(ints))

Branch prediction can hide the memory latency, i.e. it can keep the CPU working while the data is still being fetched. But this will work only if the memory bus is not saturated. If this happens, then the performance will suffer, as we have just seen.

48.2.7 Final Words

The lessons learned in this post:

- It is good idea to put the commonly used data inside the same cache line. If the data we are accessing is split between two cache lines, this has a negative effect on software performance.
- In the codes with a long dependency chains on memory loads (think pointer chasing code like traversing linked lists and trees), interleaving additional work can help hide the memory latency.
- If a code is performing random memory accesses on a large data structure, it will likely suffer from TLB cache misses. Enabling large memory pages can help mitigate the issue.
- A program accessing data with a power of two stride will suffer from conflict cache misses, a situation where useful data is evicted too early from the cache.
- Vectorization increases the pressure on the memory subsystem and limits the available speed up. Therefore, only arithmetically intense codes actually benefit from vectorization
- Branch prediction increases the pressure on the memory subsystem, but on a single core system results in performance improvement in the presence of many data cache misses. However, the price to pay is the increase in the total data volume transferred between the CPU and the memory.

In the following, last post of the series, we investigate the behavior of the memory subsystem in multi-threaded environment. Multithreading should in principle make software faster because the work is divided among several CPU cores, but in reality, the memory subsystem is one of the limiting factors in multithreading

Chapter 49

the sad state of debug performance in c++

👤 Vittorio 📅 2022-9-26 💬 ★★★

By now, it should (hopefully) be common knowledge that the term “zero cost abstraction” is a *lie*. To be fair, it’s more of a misnomer - had the term been “abstraction likely to result in zero runtime overhead after optimizations” then it would have been much more honest, but I can see why that didn’t fly…

Most C++ developers tend to accept the fact that “zero cost abstractions” provide zero runtime overhead only *with optimizations* enabled, and that they have a negative impact on compilation speed. The same developers tend to believe that the benefits of such abstractions are so valuable that having your program perform poorly in debug mode (i.e. without optimizations enabled) and compile more slowly is worth it.

I used to be one of them.

However, in the past few years, I’ve come to realize how important it is in certain domains to have performant debug builds and quick compilation. A domain where such requirements are critical is *game development*. The people working in game development tend to be quite vocal about C++’s abstractions being unfit for their work, and for good reason: games are real-time simulations that need to be playable and responsive even in debug builds - imagine the motion sickness induced by trying to debug a virtual reality game at 20FPS.

In this article, we’ll explore how C++’s abstraction model heavily relies on compiler optimizations, unveiling some unfortunate examples of unexpected performance loss. Afterwards, we will compare how the three major compilers (GCC, Clang, and MSVC) fare in this area, and we’ll discuss some potential future improvements or workarounds.

49.1 moving an int is slow

This year I gave a lightning talk at ACCU 2022 (“Moving an `int` Is Slow: Debug Performance Matters!”) with an intentionally provocative title - how can moving an `int` be *slow*?

Consider the following code:

```
#include <utility>
```

```
int main()
{
    return std::move(0);
}
```

C++ developers should know that `std::move(0)` is semantically the same as `static_cast<int&&>(0)`, and most of them would expect the compiler to generate no code for the move, even with optimizations disabled. Turns out that *GCC 12.2*, *Clang 14.0*, and *MSVC v19.x* all end up generating a `call` instruction¹ - see for yourself on Compiler Explorer.

You might be thinking that it's not a huge deal - after all, how can an extra call instruction here and there matter that much? Well, here's an example of a high-performance algorithm containing a move in its inner loop (from `libcxx`, cleaned up a bit):

```
1 template <class InputIterator, class _Tp>
2 inline constexpr
3 T accumulate(InputIterator first, InputIterator last, T init)
4 {
5     for (; first != last; ++first)
6 #if _LIBCPP_STD_VER > 17
7         init = std::move(init) + *first;
8 #else
9         init = init + *first;
10#endif
11     return init;
12 }
```

Note how, in C++17 and above, the `init` object is moved on every iteration of the loop as an optimization. Ironically, switching from C++14 to C++17 resulted in a huge debug performance loss for programs using `std::accumulate` due to the addition of that `std::move` - imagine the overhead of a per-iteration *useless* function call in a tight loop processing objects of arithmetic type!

49.2 it gets worse...much worse

`std::move` is not an isolated case - any function that semantically is a cast ends up generating a useless `call` instruction with optimizations disabled. Here are a few more examples: `std::addressof`, `std::forward`, `std::forward_like`, `std::move_if_noexcept`, `std::as_const`, `std::to_underlying`.

Let's say you don't care at all for debug performance...well, guess what - all of the aforementioned utilities also result in a function template instantiation, which slows down compilation time. Furthermore, these "casts" they will appear as part of your call stack while debugging, making the process of stepping through your code much more painful and noisy.

Utility functions that behave as casts are not the only category of abstractions that needlessly behave poorly without optimizations - you will also encounter the same issues with conceptually lightweight types such as `std::vector<T>::iterator`: no one wants to step into `iterator::operator*` and `iterator::operator++`

¹GCC and Clang have made improvements here that we will discuss later in the article.

while debugging, and no one wants the overhead of a function call on every iteration while looping over a `std::vector`. Yet, in debug mode, it happens.

You can find examples like these everywhere in C++. Notably, here's a tweet by Chris Green on `std::byte`:

Searching for `std::byte` ...

2489599 source files searched.

185 matches found.

As you can see from the linked Compiler Explorer example, the generated assembly for the bitwise shift operators on `std::byte` is dreadful, resulting in a `call` instruction for possibly the simplest and fastest operation that a CPU can perform. Of course, using a `char` would not produce such horrible assembly even with optimizations completely disabled.

49.3 what are the consequences?

The consequences of these inefficiencies are *devastating* for the reputation and usefulness of C++ in the game development world, and also (in my opinion) result in lower productivity and more debugging cycles.

- First of all, everything we have shown so far means that any game developer working on a non-toy project will not use any “zero cost abstraction”. `std::move`, `std::forward`, et cetera will all be replaced either by casts or macros.
- The use of `std::vector<T>` will be discouraged in favour of `T*`, or at the very least iteration will be done through pointers (i.e. via `std::vector<T>::data`) and not through iterators.
- Anything from the `<algorithm>` and `<numeric>` headers will likely not be used, due to the risk of major overhead (such as in `std::accumulate`) or due to the fact that those headers are notoriously heavy on compilation times.
- Safer alternatives to C types such as `std::byte` will not be used, reducing type safety and expressiveness.

Every time a seasoned C++ programmer proposes the use of a safer, harder to misuse abstraction to a game developer, they will not listen - they cannot afford to do so. Therefore people working in other domains will see game developers as primitive life forms that have not yet discovered abstractions and that like to

play with fire by juggling pointers and using macros, completely failing to see the reasons that led them to those techniques.

On the other hand, game developers will laugh at and shun C++ developers who embrace high-level abstractions and type safety, because they fail to realize that debug performance and compilation speeds might not be as important as cleaner, safer, and more maintainable code. It sucks.

I also don't have any proof of this, but I suspect that writing low-level code with the desire of optimizing the debugging experience ironically ends up increasing the frequency of debugging.

Hear me out: if someone is constantly avoiding abstractions that could make their code safer, they will inevitably write bugs more often, which will require them to debug more frequently. The debugger will be praised once the bug is resolved, therefore the developer will be more motivated to keep debug performance high by writing low-level code. It's a vicious cycle!

49.4 using optimizations in debug mode

I know what you have been thinking this whole time - you are thinking that these developers are incompetent because they could have been using `-Og` all along!

You are wrong.

First of all, `-Og` is available *only* on GCC. Clang accepts the flag, but it's exactly the same as `-O1` - LLVM maintainers have never implemented a proper debug-friendly optimization level. Even worse, MSVC has no equivalent of `-Og` whatsoever, and most game developers use MSVC as their main compiler!

Even if `-Og` was ubiquitous, it is still suboptimal to `-O0`: it can still inline code a bit too aggressively for an effective debugging session.

Any other optimization level higher than `-Og` is going to result in a very poor debugging experience due to the aggressive optimizations compilers will carry out.

49.5 what can be done?

There are a few areas where improvements could be made: (1) the *language* itself, (2) the *compilers*, and (3) the *standard libraries* - let's see.

1. We could argue that function templates are the wrong model to create lightweight abstractions over casts and bitwise operations. We could make a similar argument for class templates and lightweight types such as `std::vector<T>::iterator`.

There have been attempts in the past at introducing a language feature for “hygenic macros” that would have solved the problems described in this article, notably Jason Rice’s P1221: “Parametric Expressions” proposal. That paper hasn’t received any update in a few years, unfortunately.

Even if we managed to figure out a way to introduce “hygenic macros” into the language, it would not help with existing utilities that have been standardized as function and class templates in the past - i.e. it wouldn’t help `std::move` get any better. Maybe some sort of attribute or backwards-compatible keyword akin to `[[no_unique_address]]` combined with `[[gnu::always_inline]]` could be invented to force compilers to always inline marked functions and not require any code to be generated for them.

I have nothing concrete in mind, but it would be a nice area to explore.

2. Compilers could be a lot smarter about the way they handle these functions. And they are becoming so!

GCC 12.x introduced a new `-ffold-simple-inlines` flag as a result of my #104719 bug report, which permits the C++ frontend to fold calls to `std::move`, `std::forward`, `std::addressof`, and `std::as_const`. The documentation mentions that it should be enabled by default, but I couldn't get the compiler to perform the folding unless I manually specified the flag - see an example on Compiler Explorer.

Clang 15.x, also motivated by my #53689 issue, also introduced a similar folding pass for the same functions chosen by GCC (plus `std::move_if_noexcept`, which I assume GCC maintainers forgot about). This one seems to be enabled by default - see a comparison between Clang 14.x and Clang 15.x on Compiler Explorer.

MSVC has not yet provided any improvement in this area. The MSVC developer team has reached out after this article was published, and was very keen to improve the situation. In fact, they wrote a blog post about their improvements here. Thank you so much!

I have to mention how happy I am to see GCC and Clang maintainers stepping up to improve the debug performance situation and I deeply thank them for that. Kudos!

Regardless, I don't think an hardcoded set of functions is the right solution here. I am in favour of compilers performing some magic, but the criteria should be a bit more general.

For example, they could perform this sort of folding on any function that consists of a single `return` statement containing only a cast. And then maybe they could relax the rule to any function containing a single "basic" operation, with the intention to also cover `std::byte` and `std::vector<T>::iterator`. It would be really cool and beneficial to see something like that!

3. Finally, the standard library implementations themselves could be a little bit more clever and user-friendly.

As an example, they could use `static_cast<T&&>(x)` instead of `std::move(x)` inside the body of algorithms such as `std::accumulate`. Also, they could mark simple wrapper functions as `[[gnu::always_inline]]` or an equivalent builtin attribute forcing the compiler to inline them.

Unfortunately, the `libc++` maintainers did not really like these ideas - see this comment. I believe their arguments were quite weak, and that I made my point quite effectively on the GitHub issue, but they didn't budge.

I'd like to see some work on this area - maybe replacing a few `std::move` and `std::forward` calls with casts and adding a few attributes in strategic places could really end up benefitting the whole C++ community. Is the very minor readability hit in an already completely unreadable codebase really a worthwhile reason to not make these changes? I think not. :)

49.6 faq

- People should just write better code with less bugs, then they wouldn't have to debug it!
- Maybe…:) However, debuggers are not only used to figure out why a defect is happening, but for other reasons as well. For example, some people use debuggers to navigate through unfamiliar code, or figuring out logic bugs that sanitizers and/or abstractions cannot help with.

- Cannot the people affected by this issue selectively only compile some files without optimization?
- This is technically possible, but quite hard to achieve in practice. First of all, you don't always know where you need to look if you are debugging - you could probably make an educated guess and only disable optimizations in a few related modules, but you might not be correct and waste time.

Also, many build systems might not easily support this sort of per-file basis optimization flag settings. I can imagine it might be very difficult to integrate this idea in older codebases or proprietary/legacy build systems.

Finally, don't forget that we also get side benefits such as faster compilation by tackling this issue directly and not working around it.

49.7 conclusion

I hope you have found this exploration of C++ debug performance interesting and enlightening, and I also hope to have inspired you to contribute in this area. Feel free to reach out via email or on Twitter. to ask any question or provide your feedback/criticism!

Shameless self-promotion time!

- My book “Embracing Modern C++ Safely” is now available on all major resellers. Please consider purchasing it and share the news with your friends and colleagues - that helps a lot!
 - For more information, read the following interview: “Why 4 Bloomberg engineers wrote another C++ book”
- I offer 1-1 C++ mentoring and consulting sessions in my spare time. If it's something you are interested in, feel free to reach out at [mail \(at\) vittorioromeo \(dot\) com](mailto:vittorioromeo@vittorioromeo.com) or on Twitter.
- If you are a fan of fast-paced open-source arcade games that allow user-created content, check out Open Hexagon, my first fully-released game available on Steam and on itch.io.
 - Open Hexagon is a spiritual successor to the critically acclaimed Super Hexagon by Terry Cavanagh. Terry fully supports my project! - thank you!
- Got a VR headset and want to experience Quake, the timeless classic from 1996, as a first-class virtual reality experience? Check out Quake VR, my “labour of love” one-man project that turns Quake into an experience with many VR-only features and interactions.

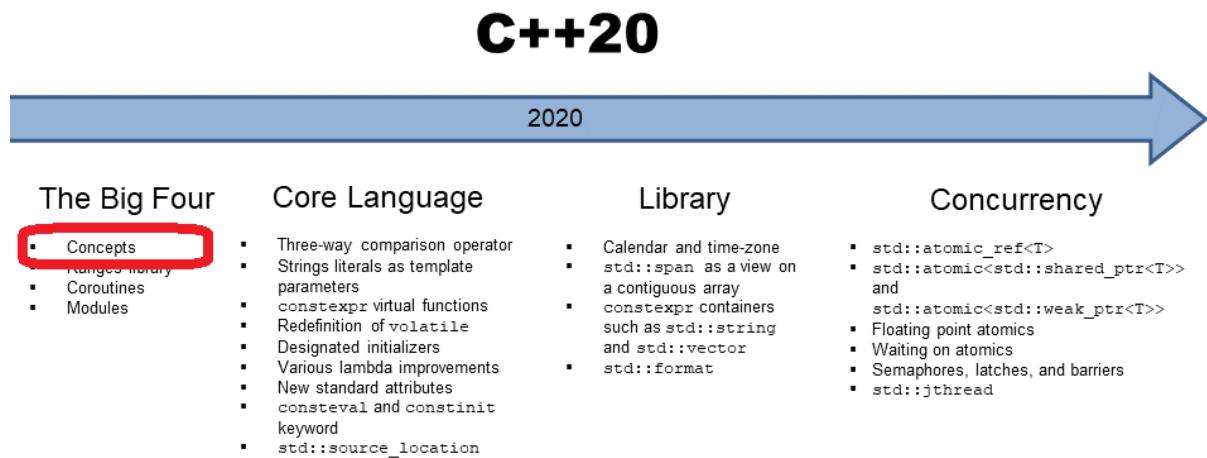
A video is worth a thousand words: YouTube: Quake VR - Release Trailer (v0.0.5).

Chapter 50

Using Requires Expression in C++20 as a Standalone Feature

▀ Rainer Grimm 📅 2022-5-2 🔒 ★★★

In my last post "Defining Concepts with Requires Expressions", I exemplified how you can use requires expressions to define concepts. Requires expressions can also be used as a standalone feature when a compile-time predicate is required.



Typical use-cases for compile-time predicates are `static_assert`, `constexpr if`, or a requires clause. A compile-time predicate is an expression that returns at compile time a boolean. Let me start this post with C++11.

50.1 `static_assert`

`static_assert` requires a compile-time predicate and a message displayed when the compile-time predicate fails. With C++17, the message is optional. With C++20, this compile-predicate can be a requires expression.

1 // staticAssertRequires.cpp

```

2
3 #include <concepts>
4 #include <iostream>
5
6 struct Fir {           // (4)
7     int count() const {
8         return 2020;
9     }
10 };
11
12 struct Sec {
13     int size() const {
14         return 2021;
15     }
16 };
17
18 int main() {
19
20     std::cout << '\n';
21
22     Fir fir;
23     static_assert(requires(Fir fir){ { fir.count() } -> std::convertible_to<int>; });    // (1)
24
25     Sec sec;
26     static_assert(requires(Sec sec){ { sec.size() } -> std::convertible_to<int>; });    // (2)
27
28     int third;
29     static_assert(requires(int third){ { third.size() } -> std::convertible_to<int>; }); // (3)
30
31     std::cout << '\n';
32 }
33 }
```

The requires expressions (lines 1, 2, and 3) check if the object has a member function `count` and its result is convertible to `int`. This check is only valid for the class `Fir` (lines 4). On the contrary, the checks in lines (2) and (3) fail.

Maybe, you want to compile code depending on a compile-time check. In this case, the C++17 feature `constexpr if` combined with requires expressions provides you the necessary tool.

50.2 `constexpr if`

`constexpr if` allows it to compile source code conditionally. For the condition, the requires expression comes into play. All branches of the if statement have to be valid.

```
rainer@seminar:~> g++ -std=c++20 staticAssertRequires.cpp -o staticAssertRequires
staticAssertRequires.cpp: In function ‘int main()’:
staticAssertRequires.cpp:26:44: error: ‘struct Sec’ has no member named ‘count’
  26 |     static_assert(requires(Sec sec){ { sec.count() } -> std::convertible_to<int>; });
                  ^~~~~~
staticAssertRequires.cpp:29:48: error: request for member ‘count’ in ‘third’, which is of non-class type ‘int’
  29 |     static_assert(requires(int third){ { third.count() } -> std::convertible_to<int>; });
                  ^~~~~~
rainer@seminar:~>
```

Thanks to `constexpr if`, you can define functions that inspect their arguments at compile time and generated different functionality based on their analysis.

```
1 // constexprIfRequires.cpp
2
3 #include <concepts>
4 #include <iostream>
5
6 struct First {
7     int count() const {
8         return 2020;
9     }
10 };
11
12 struct Second {
13     int size() const {
14         return 2021;
15     }
16 };
17
18 template <typename T>
19 int getNumberOfElements(T t) {
20
21     if constexpr (requires(T t){ { t.count() } -> std::convertible_to<int>; }) { // (1)
22         return t.count();
23     }
24     if constexpr (requires(T t){ { t.size() } -> std::convertible_to<int>; }) { // (2)
25         return t.size();
26     }
27     else return 42; // (3)
28 }
29
30 int main() {
```

```

32
33     std::cout << '\n';
34
35     First first;
36     std::cout << "getNumberOfElements(first): " << getNumberOfElements(first) << '\n';
37
38     Second second;
39     std::cout << "getNumberOfElements(second): " << getNumberOfElements(second) << '\n';
40
41     int third;
42     std::cout << "getNumberOfElements(third): " << getNumberOfElements(third) << '\n';
43
44     std::cout << '\n';
45
46 }

```

Lines (1) and (2) are crucial in this code example. In line (1), the requires expressions determine if the variable `t` has a member function `count`, that returns an `int`. Accordingly, line (2) determines if the variable `t` has a member function `size`. The else statement in line (3) is applied as a fallback.

```
rainer@seminar:~> constexprIfRequires
getNumberOfElements(first): 2020
getNumberOfElements(second): 2021
getNumberOfElements(third): 42
rainer@seminar:~>
```

50.3 Requires Clause

First of all, I have to answer the question: What is a requires clause?

There are essentially four ways to use a concept such as `std::integral`.

```

1 // conceptsIntegralVariations.cpp
2
3 #include <concepts>

```

```

4  #include <type_traits>
5  #include <iostream>
6
7  template<typename T> // (1)
8  requires std::integral<T>
9  auto gcd(T a, T b) {
10     if( b == 0 ) return a;
11     else return gcd(b, a % b);
12 }
13
14 template<typename T> // (2)
15 auto gcd1(T a, T b) requires std::integral<T> {
16     if( b == 0 ) return a;
17     else return gcd1(b, a % b);
18 }
19
20 template<std::integral T> // (3)
21 auto gcd2(T a, T b) {
22     if( b == 0 ) return a;
23     else return gcd2(b, a % b);
24 }
25
26 auto gcd3(std::integral auto a, // (4)
27             std::integral auto b) {
28     if( b == 0 ) return a;
29     else return gcd3(b, a % b);
30 }
31
32 int main(){
33
34     std::cout << '\n';
35
36     std::cout << "gcd(100, 10)= " << gcd(100, 10) << '\n';
37     std::cout << "gcd1(100, 10)= " << gcd1(100, 10) << '\n';
38     std::cout << "gcd2(100, 10)= " << gcd2(100, 10) << '\n';
39     std::cout << "gcd3(100, 10)= " << gcd3(100, 10) << '\n';
40
41     std::cout << '\n';
42
43 }
```

Thanks to the header `<concepts>`, I can use the concept `std::integral`. The concept is fulfilled if `T` is `integral`. The function name `gcd` stands for the greatest-common-divisor algorithm based on the Euclidean

algorithm.

Here are the four ways to use concepts:

1. Requires clause (line 1)
2. Trailing requires clause (line 2)
3. Constrained template parameter (line 3)
4. Abbreviated function template (line 4)

For simplicity reasons, each function template returns `auto`. There is a semantic difference between the function templates `gcd`, `gcd1`, `gcd2`, and the function `gcd3`. In the case of `gcd`, `gcd1`, or `gcd2`, arguments `a` and `b` must have the same type. This does not hold for the function `gcd3`. Parameters `a` and `b` can have different types but must both fulfill the concept `std::integral`.

```
rainer@seminar:~> conceptsIntegralVariations

gcd(100, 10)= 10
gcd1(100, 10)= 10
gcd2(100, 10)= 10
gcd3(100, 10)= 10

rainer@seminar:~>
```

The functions `gcd` and `codegcd1` use requires clauses.

There is an interesting fact about requires clauses. You can use any compile-time predicate as an expression. I check in the following requirces clause, if an int as a non-type template parameter is smaller than 20.

```
1 // requiresClause.cpp
2
3 #include <iostream>
4
5 template <unsigned int i>
6 requires (i <= 20)           // (1)
7 int sum(int j) {
8     return i + j;
9 }
```

```

11 int main() {
12
13     std::cout << '\n';
14
15     std::cout << "sum<20>(2000): " << sum<20>(2000) << '\n',
16     // std::cout << "sum<23>(2000): " << sum<23>(2000) << '\n', // ERROR
17
18     std::cout << '\n';
19
20 }
```

The compile-time predicate used in line (1) exemplifies an interesting point: the requirement is applied to the non-type `i`, and not on a type as usual.



```
sum<20>(2000): 2020
```

When you use the commented-out line in the `main` program, the clang compiler reports the following error:

```

<source>:17:39: error: no matching function for call to 'sum'
    std::cout << "sum<23>(2000): " << sum<23>(2000) << '\n', // ERROR
                                         ^~~~~~
<source>:7:5: note: candidate template ignored: constraints not satisfied [with i = 23]
int sum(int j) {
    ^
<source>:6:11: note: because '23U <= 20' (23 <= 20) evaluated to false
requires (i <= 20)
    ^
```

Here are more details about non-type template parameters: “Alias Templates and Template Parameters”.

Typically, you use a concept in a `requires` clause, but there is more: `requires requires` or anonymous concepts

50.4 ‘requires requires’ or anonymous concepts

You can define an anonymous concept and directly use it. In general, you should not do it. Anonymous concepts make your code hard to read, and you cannot reuse your concepts.

```

template<typename T>
requires requires (T x) { x + x; }
auto add(T a, T b) {
    return a + b;
}
```

The function template defines its concept ad-hoc. The function template `add` uses a requires expression (`requires(T x) { x + x; }`) inside a requires clause. The anonymous concept is equivalent to the following concept `Addable`.

```
template<typename T>
concept Addable = requires (T a, T b) {
    a + b;
};
```

Consequentially, the following four implementations of the function template `add` are equivalent to the previous one:

```
1 template<typename T> // requires clause
2 requires Addable<T>
3 auto add(T a, T b) {
4     return a + b;
5 }
6
7 template<typename T> // trailing requires clause
8 auto add(T a, T b) requires Addable<T> {
9     return a + b;
10 }
11
12 template<Addable T> // constrained template parameter
13 auto add(T a, T b){
14     return a + b;
15 }
16             // abbreviated function template
17 auto add(Addable auto a, Addable auto b) {
18     return a + b;
19 }
```

As a short reminder: The last implementation based on abbreviated function templates syntax can deal with values having different types.

I want to emphasize it once more: **Concepts should encapsulate general ideas and give them a self-explanatory name for reuse. They are invaluable for maintaining code. Anonymous concepts read more like syntactic constraints on template parameters and should be avoided.**

Chapter 51

What is faster: `vec.emplace_back(x)` or `vec[x]` ?

👤 Ivica Bogosavljević 📅 2022-8-24 💬 ★★★

We at *Johnny's Software Lab LLC* are experts in performance. If performance is in any way concern in your software project, feel free to contact us.

When we need to fill `std::vector` with values and the size of vector is known in advance, there are two possibilities: using `emplace_back()` or using `operator[]`. For the `emplace_back()` we should reserve the necessary amount of space with `reserve()` before emplacing into vector. This will avoid unnecessary vector regrow and benefit performance. Alternatively, if we want to use `operator[]`, we need to initialize all the elements of the vector before using it.

A common wisdom says that `emplace_back()` version should be faster. When using `emplace_back()`, we are accessing the elements of the vector only once, when inserting values into vector. On the other hand, when using `operator[]`, the CPU will need the run through the vector two times: first time when the vector is constructed, and the second time when we are actually filling the vector.

51.1 The Experiment

To test the behavior of these two approaches, we conduct an experiment. In the experiment, we are limiting ourselves to a vector of doubles with 256M doubles and 2 GB size. We have a following really simple code for the `emplace_back()` version:

```
1 std::vector<double> result;
2 result.reserve(in.size());
3 size_t size = in.size();
4 for (size_t i = 0; i < size; i++) {
5     result.emplace_back(std::sqrt(in[i]));
6 }
```

And here is the corresponding code for the `operator[]` version:

```

1 std::vector<double> result(in.size());
2 size_t size = in.size();
3 for (size_t i = 0; i < size; i++) {
4     result[i] = std::sqrt(in[i]);
5 }
```

We are measuring the runtime of the hot loop, as well as the total runtime (including vector initialization). Let's see if our assumptions about the runtime verify in tests.

51.2 First Results

Here are the first results¹:

	emplace_back()	operator[]
Total	Runtime: 0.915 s	Runtime: 0.815 s
Hot loop	Runtime: 0.915 s	Runtime: 0.276 s

Unexpectedly, the total runtime for the operator[] version is about 10% faster. But the numbers look weird. There are a few interesting questions about these numbers: (1) why is the total runtime for the operator[] version faster? (2) why is the hot loop runtime for the operator[] 3.2 times faster than emplace_back() hot loop and (3) in operator[] why is the initialization so much slower than the hot loop (0.539 s vs 0.276 s)²?

When your debugging effort comes to a standstill and you do not know what to measure, a good approach is to measure everything, then compare the results to try to figure out what's going on. It is also useful to start to answer the simplest question first and then build on top of that.

51.3 Question 1: Why is vector initialization so slow for operator[] ?

The operator[] hot loop runtime is 0.276 s, whereas the vector initialization runtime is 0.539 s. What this means is that initializing the vector was almost two times slower than the hot loop, but this doesn't make sense. Vector initialization is a simple memset, the simplest of all. In the hot loop we have sqrt which is not a cheap operation. So a natural expectation would be for the runtimes to be reversed – initialization faster and sqrt slower. To understand what is going on, we need to investigate further.

Let's look at the instruction count and the CPI (cycles-per-instruction) metric for emplace back:

	Instructions	CPI
Total	336.08 M	5.124
Hot loop	335.55 M	m2.718

These numbers don't make even more sense. The vector initialization was executing 99% of total instructions, yet it is faster than the 1 % of instructions executed for the vector initialization.

When things like this happen, we are essentially stuck. So, we measure everything. To do this, we are using LIKWID, a tool that can collect hardware counters. In addition, we are using LIKWID wrapper that can measure things like user mode runtime, system mode runtime, pagefaults, etc.

After the measurements, the most noticeable difference between the two codes is the time spent in system mode and the number of page faults. Here are the numbers:

¹Average results, repeated 10 times. The standard deviation for all the experiments was very small and therefore not reported.

²里缪注: Total = vector initialization runtime + Hot loop runtime

51.4. QUESTION 2: WHY IS THE HOT LOOP IN `EMPLACE_BACK()` SLOWER? PAGEFAULTS OR SOMETHING ELSE? 57

	<code>emplace_back()</code>	<code>operator[]</code>
Total	User: 0.542s System: 0.371s Pagefaults: 524289	User: 0.366s System: 0.454s Pagefaults: 524289
Hot loop	User: 0.542s System: 0.371s Pagefaults: 524288	User: 0.278s System: 0.000s Pagefaults: 0

These numbers are very revealing. First access to the uninitialized memory allocated by the vector will result in many minor pagefaults. Minor pagefaults happen when a page in virtual memory is not yet backed up by a page in physical memory. When a minor pagefault happens, the kernel takes over to perform the allocation. Switching to kernel and allocating memory is relatively expensive process.

For the `emplace_back()` version, physical memory allocations happen inside the hot loop, because our test program is accessing memory for the first time inside the hot loop. For this reason we see many pagefaults and large amount of time spent in the system mode. **For the `operator[]` version, the physical memory allocation happens when the vector is constructed for the first time.** Therefore, the vector initialization takes so much time. In the hot loop there is no physical memory allocation for the `operator[]` version; that's why its hot loop is so much faster compared to `emplace_back()` hot loop.

We still don't understand why the hot loop in `emplace_back()` is slower. Is it only because of the page faults, or something else.

51.4 Question 2: Why is the hot loop in `emplace_back()` slower? Page-faults or something else?

To discover this, we need to repeat the same test, but this time `emplace_back()` needs to work with virtual memory that has already been allocated in physical memory. To achieve this, we need to overwrite the memory allocator for `std::vector`.

We have created `malloc_wrapper` and passed it to `std::vector`. When memory is allocated through `malloc_wrapper`, it is also initialized to some default value. By doing this, pagefaults move from `emplace_back()` to `reserve()` method.

Here are the runtimes, total instruction count and CPI for the two hot loops:

	<code>emplace_back()</code>	<code>emplace_back()</code> with preinitialized memory
Runtime	0.276 s	0.496 s
Instructions	335.548 M	2952.794 M
CPI	2.72	0.606
Pagefaults	0	0
Systime	0 s	0 s

The `emplace_back()` version is slower about two times, but what surprises is the instruction count. The `operator[]` version executes almost ten times less instructions, albeit with much lower efficiency (CPI 2.72 vs 0.606).

We do expect that `emplace_back()` executes more instruction, because it needs to check for array boundaries in each iteration and regrow the array if necessary. But this doesn't account for 10 times more instructions.

The reason for this large difference in instruction count is typically vectorization: the compiler can emit special vector instructions, which process four doubles in a single instruction. In addition, the `operator[]`

hot loop is without conditionals. We can verify this in two ways: (1) compiler optimization report or (2) looking at disassembly.

We are using CLANG for this experiment, and passing `-Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize` will provide us with the vectorization report. **The results are clear: hot loop in `operator[]` is vectorized; in contrast hot loop in `emplace_back()` isn't.**

51.5 Question 3: Why is only the `operator[]` hot loop vectorized?

The `operator[]` hot loop is vectorized. The `emplace_back()` hot loop isn't. The reason, according to the CLANG vectorization report is loop not vectorized: could not determine number of loop iterations.

The vectorization report indicates that the number of iterations cannot be calculated before the loop starts, but this is not the case. The loop has exactly 256M iterations, so the report is misleading. The real reason is how `emplace_back()` is implemented!

Inside `emplace_back()`, two things happen. The code first checks if the size of vector is big enough to hold the next value to insert. If it is not, it makes another call to the system allocator to request a larger buffer, copies the values from the old buffer to the new buffer and destroys the old buffer. After this, the vector is large enough to hold the new value.

When we make a call to `reserve()`, we know for sure there will be no call to the system allocator, data copying and destruction when `emplace_back()` is called. But the compiler doesn't know this. So, the compiler must emit a check if the vector is large enough, and grow the vector if it isn't. **But just having a conditional call to a function, even if the call is never made, is a vectorization inhibitor. The compiler must emit the slower scalar version for calls to `push_back()` and `emplace_back()`, even if we know for sure that there will always be enough place in the vector.**

51.6 Question 4: What is the effect of pagefaults on runtime for the two hot loops?

To understand the effect of pagefaults on runtime, we need to compare both hot loops with and without pagefaults. For the `emplace_back()` version, we already have two implementations, one with page faults, one without. For the `operator[]` version, we only have implementation without pagefaults. We can emulate the implementation with pagefaults by using C style array instead of vector. Internally, vectors use C style arrays, so even if the code is not the same, the compiler emits very similar assembly.

Here is the relevant data:

	Without pagefaults	With pagefaults
Hot loop in <code>emplace_back()</code>	Runtime: 0.496 s Instructions: 2952.794 M Pagefaults: 0	Runtime: 0.920 s Instructions: 3221.755 M Pagefaults: 524288
Hot loop in <code>operator[]</code>	Runtime: 0.276 s Instructions: 335.55 M Pagefaults: 0	Runtime: 0.714 s Instructions: 336.07 M Pagefaults: 524288

If we look at these numbers, we see that **the slowdown due to pagefaults is constant**, 0.424 s for the `emplace_back()` and 0.438 s for the `operator[]`. This is the time needed to resolve 524288 pagefaults,

which when multiplied by page size of 4 kB, results in exactly 2 GB of data. The time needed per pagefault is about 0.8 µs.

The slowdown due to pagefaults is more pronounced on the vectorized version of the code, since it is faster and more efficient.

51.7 Question 5: If the operator[] hot loop is not vectorized, which version is faster then?

Originally, we established that the operator[] version is slightly faster, and this is because its hot loop is vectorized. But what happens if compiler omits vectorization. Which version is faster than? Here are the numbers:

	<code>emplace_back()</code>	<code>operator[]</code> without vectorization
Total	Runtime: 0.920 s Instructions: 3221.76 M Pagefaults: 524289	Runtime: 1.031 s Instructions: 1611.15 M Pagefaults: 524289
Hot loop	Runtime: 0.920 s Instructions: 3221.75 M Pagefaults: 524288	Runtime: 0.491 s Instructions: 1610.61 M Pagefaults: 0

Without vectorization, the `emplace_back()` version is faster, even though it executes almost two times more instructions.

51.8 Discussion

We picked this example intentionally to show how deep the rabbit hole goes. What apparently seems to be a very trivial question, “which version is faster” translates into a much more complex question involving kernel space, compiler optimizations and hardware efficiency.

A question you might ask is what happens if the vector holds other types or it is running on other architectures. In principle, if the type stored in the vector is a simple type (not a class or a struct), the loop is often vectorizable and one could expect to see similar numbers as we have seen here. With larger classes, it is the opposite: vectorization doesn’t pay off because of the low memory efficiency, so the operator[] version will in principle slower. But, as always with software performance, the runtime numbers are the ones that give the final verdict.

If you are interested in this sort of performance optimizations, I deeply recommend to read about the essence of vectorization here, and about page faults here.

Chapter 52

P2723R0 Zero-initialize objects of automatic storage duration

👤 JF Bastien (Woven Planet) 📅 2022-11-15 🏷️ ★★★

This version: <http://wg21.link/P2723> Author: JF Bastien (Woven Planet) Audience: EWG, SG12, SG22 Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language —C++ Source: github.com/jfbastien/papers/blob/master/source/P2723r0.bs

52.1 Summary

Reading uninitialized values is undefined behavior which leads to well-known security issues [CWE-457], from information leaks to attacker-controlled values being used to control execution, leading to an exploit. This can occur when:

- The compiler re-uses stack slots, and a value is used uninitialized;
- The compiler re-uses a register, and a value is used uninitialized;
- Stack structs / arrays / unions with padding are copied; and
- A heap-allocated value isn't initialized before being used.

This proposal only addresses the above issues when they occur for automatic storage duration objects (on the stack). The proposal does not tackle objects with dynamic object duration. Objects with static or thread local duration are not problematic because they are already zero-initialized if they cannot be constant initialized, even if they are non-trivial and have a constructor execute after this initialization.

```
1 static char global[128];           // already zero-initialized
2 thread_local int thread_state;    // already zero-initialized
3
4 int main() {
5     char buffer[128];             // not zero-initialized before this proposal
6 }
```

```

7   struct {
8     char type;
9     int payload;
10    } padded; // padding zero-initialized with this proposal
11
12   union {
13     char small;
14     int big;
15   } lopsided = '?'; // padding zero-initialized with this proposal
16
17   int size = getSize();
18   char vla[size]; // in C code, VLAs are zero-initialized with this proposal
19
20   char *allocated = malloc(128); // unchanged
21   int *new_int = new int; // unchanged
22   char *new_arr = new char[128]; // unchanged
23 }
```

This proposal therefore transforms some runtime undefined behavior into well-defined behavior.

Adopting this change would mitigate or extinguish around 10% of exploits against security-relevant codebases (see numbers from [AndroidSec] and [InitAll] for a representative sample, or [KCC] for a long list).

We propose to zero-initialize all objects of automatic storage duration, making C++ safer by default.

This was implemented as an opt-in compiler flag in 2018 for LLVM [LLVMReview] and MSVC [WinKernel], and 2021 in for GCC [GCCReview]. Since then it has been deployed to:

- The OS of every desktop, laptop, and smartphone that you own;
- The web browser you’re using to read this paper;
- Many kernel extensions and userspace program in your laptop and smartphone; and
- Likely to your favorite videogame console.

Unless of course you like esoteric devices.

The above codebases contain many millions of lines of code of C, C++, Objective-C, and Objective-C++.

The performance impact is negligible (less than 0.5% regression) to slightly positive (that is, some code gets *faster* by up to 1%). The code size impact is negligible (smaller than 0.5%). Compile-time regressions are negligible. Were overheads to matter for particular coding patterns, compilers would be able to obviate most of them.

The only significant performance/code regressions are when code has very large automatic storage duration objects. We provide an attribute to opt-out of zero-initialization of objects of automatic storage duration. We then expect that programmer can audit their code for this attribute, and ensure that the unsafe subset of C++ is used in a safe manner.

This change was not possible 30 years ago because optimizations simply were not as good as they are today, and the costs were too high. The costs are now negligible. We provide details in § 5 Performance.

Overall, this proposal is a targeted fix which stands on its own, but should be considered as part of a wider push for type and resource safe C++ such as in [P2687r0].

52.2 Opting out

In deploying this scheme, we've found that some codebases see unacceptable regressions because they allocate large values on the stack and see performance / size regressions. We therefore propose standardizing an attribute which denotes intent, `[[uninitialized]]`. When put on automatic variables, no initialization is performed. The programmer is then responsible for what happens because they are back to C++23 behavior. This makes C++ safe by default for this class of bugs, and provides an escape hatch when the safety is too onerous.

An `[[uninitialized]]` attribute was proposed in [P0632r0].

As a quality of implementation tool, and to ease transitions, some implementations might choose to diagnose on large stack values that might benefit from this attribute. We caution implementations to avoid doing so in their front-end, but rather to do this as a post-optimization annotation to reduce noise. This was done in LLVM [AutoInitSummary].

52.3 Security details

What exactly is the security issues?

Currently, uninitialized stack variables are Undefined Behavior if they are used. The typical outcome is that the program reads stale stack or register values, that is whatever previous value happened to be compiled to reside in the same stack slot or register. This leads to bugs. Here are the potential outcomes:

- *Benign*: in the best case the program causes an unexpected result.
- *Exploit*: in the worst case it leads to an exploit. There are three outcomes that can lead to an exploit:
 - *Read primitive*: this exploit could be formed from reading leaked secrets that reside on the stack or in registers, and using this information to exploit another bug. For example, leaking an important address to defeat ASLR.
 - *Write primitive*: an uninitialized value is used to perform a write to an arbitrary address. Such a write can be transformed into an execute primitive. For example a stack slot is used on a particular execution path to determine the address of a write. The attack can control the previous stack slot's content, and therefore control where the write occurs. Similarly, an attacker could control which value is written, but not where it is written.
 - *Execute primitive*: an uninitialized value is used to call a function.

Here are a few examples:

```

1 int get_hw_address(struct device *dev, struct user *usr) {
2     unsigned char addr[MAX_ADDR_LEN];           // leak this memory

```

```

3
4     if (!dev->has_address)
5         return -EOPNOTSUPP;
6
7     dev->get_hw_address(addr);           // if this doesn't fill addr
8     return copy_out(usr, addr, sizeof(addr)); // copies all
9 }
```

Example from [LinuxInfoleak]. What can this leak? ASLR information, or anything from another process. ASLR leak can enable Return Oriented Programming (ROP) or make an arbitrary write effective (e.g. figure out where high-value data structures are). Even padding could leak secrets.

```

1 int queue_manage() {
2     struct async_request *backlog;           // uninitialized
3
4     if (engine->state == IDLE)             // usually true
5         backlog = get_backlog(&engine->queue);
6
7     if (backlog)
8         backlog->complete(backlog, -EINPROGRESS); // oops
9
10    return 0;
11 }
```

Example from [LinuxUse]. The attacker can control the value of backlog by making a previous function call, and ensuring the same stack slot gets reused.

Security-minded folks think this is a good idea. The Microsoft Windows security team [WinKernel] say:

Between 2017 and mid 2018, this feature would have killed 49 MSRC cases that involved uninitialized struct data leaking across a trust boundary. It would have also mitigated a number of bugs involving uninitialized struct data being used directly. To date, we are seeing noise level performance regressions caused by this change. We accomplished this by improving the compilers ability to kill redundant stores. While everything is initialized at declaration, most of these initializations can be proven redundant and eliminated.

They use pure zero initialization, and claim to have taken the overheads down to within noise. They provide more details in [CppConMSRC] and [InitAll]. Don't just trust Microsoft's Windows security team though, here's one of the upstream Linux Kernel security developer asking for this [CLessDangerous], and Linus agreeing [Linus]. [LinuxExploits] is an overview of a real-world execution control exploit using an uninitialized stack variable on Linux. It's been proposed for GCC [init-local-vars] and LLVM [LocalInit] before.

In Chrome we find 1100+ security bugs for “use of uninitialized value” [ChromeUninit].

12% of all exploitable bugs in Android are of this kind according to [AndroidSec].

Kostya Serebryany has a very long list of issues caused by usage of uninitialized stack variables [KCC].

52.4 Alternatives

When this is discussed, the discussion often goes to a few places.

First, people will suggest using tools such as memory sanitizer or valgrind. Yes they should, but doing so:

- Requires compiling everything in the process with memory sanitizer.
- Can't deploy in production.
- Only find that is executed in testing.
- 3x slower and memory-hungry.

The next suggestion is usually couldn't you just test / fuzz / code-review better or just write perfect code? However:

- We are (well, except the perfect code part).
- It's not sufficient, as evidenced by the exploits.
- Attackers find what programmers don't think of, and what fuzzers don't hit.

The annoyed suggester then says "couldn't you just use `-Werror=uninitialized` and fix everything it complains about?" This is similar to the [CoreGuidelines] recommendation. You are beginning to expect shortcoming, in this case:

- Too much code to change.
- Current `-Wuninitialized` is low quality, has false positives and false negatives. We could get better analysis if we had a compiler-frontend-based IR, but it still wouldn't be noise-free.
- Similarly, static analysis isn't smart enough.
- The value chosen to initialize doesn't necessarily make sense, meaning that code might move from unsafe to merely incorrect.
- Code doesn't express intent anymore, which leads readers to assume that the value is sensible when it might have simply been there to address the diagnostic.
- It prevents checkers (static analysis, sanitizers) from diagnosing code because it was given semantics which aren't intended.

Couldn't you just use definitive initialization in the language? For example as explained in [CppConDefinite] and [CppConComplexity]. This one is interesting. Languages such as C# and Swift allow uninitialized values, but only if the compiler can prove that they're not used uninitialized, otherwise they must be initialized. It's similar to enforcing `-Wuninitialized`, depending on what is standardized. We then have a choice: standardize a simple analysis (such as "must be initialized within the block or before it escapes") and risk needless initializations (with mitigations explained in the prior references), or standardize a more complex analysis. We would likely want to deploy a low-code-churn change, therefore a high-quality analysis.

This would require that the committee standardize the analysis, lest compilers diverge. But a high-quality analysis tends to be complex, and change as it improves, which is not sensible to standardize. This would leave us standardizing a high-code-churn analysis with much more limited visibility. Further with definitive initialization, it's then unclear whether `int derp = 0;` lends meaning to `0`, or whether it's just there to shut that warning up.

An alternative for the `[[uninitialized]]` attribute is to use a keyword. We can argue about which keyword is right. However, maybe through [P0146r1], we'll eventually agree that `void` is the right name for "This variable is not a variable of honor... no highly esteemed deed is commemorated here... nothing valued is here." This approach meshes well with a definitive initialization approach because we would keep the semantics of "no value was intended on this variable".

One alternative that was suggested instead of zero-initialization is to value-initialize. This would invoke constructors and therefore be a no-go as a significant behavior change. That said, some form of "trivial enough" initialization might be an acceptable approach to increase correctness (not merely safety) for types for which zero isn't a valid value.

We could instead keep the out-of-band option that is already implemented in all major compilers. That out-of-band option is opt-in and keeps the status quo where "C++ is unsafe by default". Further, many compiler implementors are dissatisfied by this option because they believe it's effectively a language fork that users of the out-of-band option will come to rely on (this strategy is mitigated in MSVC by using a non-zero pattern in debug builds according to [InitAll]). These developers would rather see the language standardize the option officially.

Rather than zero-initializing, we could use another value. For example, the author implemented a mode where the fundamental types dictate the value used:

- Integral → repeated `0xAA`
- Pointers → repeated `0xAA`
- Floating-point → repeated `0xFF`

This is advantageous in 64-bit platforms because all pointer uses will trap with a very recognizable value, though most modern platforms will also trap for large offsets from the null pointer. Floating-point values are NaNs with payload, which propagate on floating-point operations and are therefore easy to root-cause. It is problematic for some integer values that represent sizes in a bounds check, because such a number is effectively larger than any bounds and would potentially permit out-of-bounds accesses that a zero initialization would not.

Unfortunately, this scheme optimizes much more poorly and end up being a roughly 1.5% size and performance regression on some workloads according to [GoogleNumbers] and others.

Using repeated bytes for initialization is critical to effective code generation because instruction sets can materialize repeated byte patterns much more efficiently than any larger value. Any approach other than the above is bound to have a worst performance impact. That is why choosing implementation-defined values or runtime random values isn't a useful approach.

Another alternative is to guarantee that accessing uninitialized values is implementation-defined behavior, where implementations *must* choose to either, on a per-value basis:

- zero-initialize; or

- trap (we can argue separately as to whether "trap" is immediate termination, Itanium [NaT], or `std::terminate` or something else).

Implementations would then do their best to trap on uninitialized value access, but when they can't prove that an access is uninitialized through however complex optimization scheme, they would revert to zero-initialization. This is an interesting approach which Richard Smith has implemented a prototype of in [Trap]. However, the author's and others' experience is that such a change is rejected by teams because it cannot be deployed to codebases that seem to work just fine. The Committee might decide to go with this approach, and see whether the default in various implementations is "always zero" or "sometimes trap".

We could also standardized a hybrid approach that mixes some of the above, creating implementation-defined alternatives which are effectively different build modes that the standard allows. This is what [CarbonInit] does.

52.5 Performance

Previous publications [Zeroing] have cited 2.7 to 4.5% averages. This was true because, in the author's experience implementing this in LLVM [LLVMJFB], the compiler simply didn't perform sufficient optimizations. Deploying this change required a variety of optimizations to remove needless work and reduce code size. These optimizations were generally useful for other code, not just automatic variable initialization.

As stated in the introduction, the performance impact is now negligible (less than 0.5% regression) to slightly positive (that is, some code gets *faster* by up to 1%). The code size impact is negligible (smaller than 0.5%).

Why do we sometimes observe a performance progression with zero-initialization? There are a few potential reasons:

- Zeroing a register or stack value can break dependencies, allowing more Instructions Per Cycles (IPC) in an out-of-order CPU by unblocking scheduling of instructions that were waiting on what seemed like a long critical path of instructions but were in fact only false-sharing registers. The register can be renamed to the hardware-internal zero register, and instructions scheduled.
- Zeroing of entire cache entries is magical, though for the stack this is rarely the case. There's questions of temporality, special instructions, and special logic to support zeroed cachelines.

Here are a few of the optimizations which have helped reduce overheads in LLVM:

- CodeGen: use non-zero `memset` for automatic variables D49771
- Merge clang's `isRepeatedBytePattern` with LLVM's `isBytewiseValue` D51751
- SelectionDAG: Reuse bigger constants in `memset` D53181
- ARM64: improve non-zero `memset` isel by 2x D51706
- Double sign-extend stores not merged going to stack D54846 and D54847
- LoopUnroll: support loops w/ exiting headers & uncond latches D61962
- ConstantMerge: merge common initial sequences D50593

- IPO should track pointer values which are written to, enabling DSO DSO
- Missed dead store across basic blocks pr40527

This doesn't cover all relevant optimizations. Some optimizations will be architecture-specific, meaning that deploying this change to new architectures will require some optimization work to reduce cost.

New codebases using automatic variable initialization might trigger missed-optimizations in compilers and experience a performance or size regression. The wide deployment of the opt-in compiler flag means that this is unlikely, but were this to happen then a bug should be filed against the compiler to fix the missed optimization.

For example, here is Android's experience on performance costs [AndroidCosts].

Some of the performance costs are hidden by modern CPUs being heavily superscalar. Embedded CPUs are often in-order, code running on them might therefore have a performance regression that is roughly proportional to code size increases. Here too, compiler optimizations can significantly reduce these costs.

52.6 Caveats

There are a few caveats to this proposal.

Making all automatic variables explicitly zero means that developers will come to rely on it. The current status quo forces developers to express intent, and new code might decide not to do so and simply use the zero that they know to be there. This would then make it impossible to distinguish "purposeful use of the uninitialized zero" from "accidental use of the uninitialized zero". Tools such as memory sanitizers and valgrind would therefore be unable to diagnose correctness issues (but we would have removed the security issues). It should still be best practice to only assign a value to a variable when this value is meaningful, and only use an "uninitialized" value when meaning has been given to it.

This proposal will now mean that objects with dynamic storage duration are now uniquely different from all other storage duration objects, in that they alone are uninitialized. We could explore guaranteeing initialization of these objects, but this is a much newer area of research and deployment as shown in [XNUHeap]. The strategy for automatically initializing objects of dynamic storage durations depend on a variety of factors, there's no agreed-upon one right solution for all codebases. Each implementation approach has tradeoffs that warrant more study.

Some types might not have a valid zero value, for example an enum might not assign meaning to zero. Initializing this type to zero is then safe, but not correct. However, C++ has constructors to deal with this type of issue.

For some types, the zero value might be the "dangerous" one, for example a null pointer might be a special sentinel value saying "has all privileges" in a kernel. A concrete example is on Linux where `task->uid` of zero means `root`, uninitialized reads of zero have been the source of CVEs on Linux before.

Some people think that reading uninitialized stack variables is a good source of randomness to seed a random number generator, but these people are wrong [Randomness].

As an implementation concern for LLVM (and maybe others), not all variables of automatic storage duration are easy to properly zero-initialize. Variables declared in unreachable code, and used later, aren't currently initialized. This includes `goto`, Duff's device, other objectionable uses of `switch`. Such things should rather be a hard-error in any serious codebase.

`volatile` stack variables are weird. That's pre-existing, it's really the language's fault and this proposal keeps it weird. In LLVM, the author opted to initialize all `volatile` stack variables to zero. This is technically acceptable because they don't have any special hardware or external meaning, they're only used to block the optimizer. It's technically incorrect because the compiler shouldn't be performing extra accesses to `volatile` variables, but one could argue that the compiler is zero-initializing the storage right before the `volatile` is constructed. This would be a pointless discussion which the author is happy to have if the counterparty is buying.

52.7 Wording

Wording for this proposal will be provided once an approach is agreed upon.

The author expects that the solution that is agreed upon maintains the property that:

- Taking the address, passing this address, and manipulating the address of data not explicitly initialized remains valid.
- `memcpy` of data not explicitly initialized (including padding) remains valid, but reading from the data or the `memcpyd` data is invalid.

52.7.1 References

52.7.2 Informative References

- [AndroidCosts] Alexander Potapenko. Fighting Uninitialized Memory in the Kernel. 2020. https://clangbuiltlinux.github.io/CBL-meetup-2020-slides/glider/Fighting_uninitialized_memory_%40_CBL_Meetup_2020.pdf
- [AndroidSec] Nick Kralevich. How vulnerabilities help shape security features and mitigations in Android. 2016-08-04. <https://www.blackhat.com/docs/us-16/materials/us-16-Kralevich-The-Art-Of-De.pdf>
- [AutoInitSummary] Florian Hahn. [llvm-dev] RFC: Combining Annotation Metadata and Remarks. 2020-11-04. <https://lists.llvm.org/pipermail/llvm-dev/2020-November/146393.html>
- [CarbonInit] Chandler Carruth. Carbon: Initialization of memory and variables. 2021-06-22. <https://github.com/carbon-language/carbon-lang/blob/trunk/proposals/p0257.md>
- [ChromeUninit] multiple. Chromium code search for Use-of-uninitialized-value. multiple. <https://bugs.chromium.org/p/chromium/issues/list?can=1&q=Stability%3DMemory-MemorySanitizer+-status%3ADuplicate+-status%3AWontFix+Use-of-uninitialized-value>
- [CLessDangerous] Kees Cook. Making C Less Dangerous. 2018-08-28. <https://outflux.net/slides/2018/lss/danger.pdf>
- [CoreGuidelines] Herb Sutter; Bjarne Stroustrup. C++ Core Guidelines: ES.20: Always initialize an object. unknown. <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-always>

- [CppConComplexity] Herb Sutter. Empirically Measuring, & Reducing, C++'s Accidental Complexity. 2020-10-11. <https://www.youtube.com/watch?v=6lur0Cdaj0Y&t=971s>
- [CppConDefinite] Herb Sutter. Can C++ be 10x Simpler & Safer?. 2022-11-13. <https://www.youtube.com/clip/UgkxpWxXNaoWvGqhlpSdWRQs0M3ayQcEhDoh>
- [CppConMSRC] Joseph Bialek; Shayne Hiet-Block. Killing Uninitialized Memory: Protecting the OS Without Destroying Performance. 2019-10-13. <https://www.youtube.com/watch?v=rQWjF8NvqAU>
- [CWE-457] MITRE. CWE-457: Use of Uninitialized Variable. 2006-07-19. <https://cwe.mitre.org/data/definitions/457.html>
- [GCCReview] Qing Zhao. [RFC][patch for gcc12][version 1] add -ftrivial-auto-var-init and variable attribute "uninitialized" to gcc. 2021-02-18. <https://gcc.gnu.org/pipermail/gcc-patches/2021-February/565514.html>
- [GoogleNumbers] Kostya Serebryany. [cfe-dev] [RFC] automatic variable initialization. 2019-01-16. <https://lists.llvm.org/pipermail/cfe-dev/2019-January/060878.html>
- [INIT-LOCAL-VARS] Florian Weimer. -finit-local-vars option. 2014-06-06. <https://gcc.gnu.org/ml/gcc-patches/2014-06/msg00615.html>
- [InitAll] Joseph Bialek. Solving Uninitialized Stack Memory on Windows. 2020-05-13. <https://msrc-blog.microsoft.com/2020/05/13/solving-uninitialized-stack-memory-on-windows/>
- [KCC] Kostya Serebryany. [cfe-dev] [RFC] automatic variable initialization. 2018-11-15. <https://lists.llvm.org/pipermail/cfe-dev/2018-November/060177.html>
- [Linus] Linus Torvalds. Re: [GIT PULL] meminit fix for v5.3-rc2. 2019-06-30. https://lore.kernel.org/lkml/CAHk-=wgTM+cN7zyUZacGQDv3DuuoA4L0RNPWgb1Y_Z1p4iedNQ@mail.gmail.com/
- [LinuxExploits] Kees Cook. Kernel Exploitation via Uninitialized Stack. 2019-08. <https://outflux.net/slides/2011/defcon/kernel-exploitation.pdf>
- [LinuxInfoleak] Mathias Krause. Linux kernel: net - three info leaks in rtnl. 2013-03-19. <https://seclists.org/oss-sec/2013/q1/693>
- [LinuxUse] Kangjie Lu; et al. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. 2017-03-01. <https://www-users.cs.umn.edu/~kjlu/papers/tss.pdf>
- [LLVMJFB] JF Bastien. Mitigating Undefined Behavior. 2019-12-12. <https://www.youtube.com/watch?v=I-XUHPimq3o>
- [LLVMReview] JF Bastien. Automatic variable initialization. 2018-11-15. <https://reviews.llvm.org/D54604>
- [LocalInit] thestinger. add LocalInit sanitizer. 2018-06-22. https://github.com/AndroidHardeningArchive/platform_external_clang/commit/776a0955ef6686d23a82d2e6a3cbd4a6a882c31c

- [NaT] Raymond Chen. Uninitialized garbage on ia64 can be deadly. 2004-01-19. <https://devblogs.microsoft.com/oldnewthing/20040119-00/?p=41003>
- [P0146r1] Matt Calabrese. Regular Void. 11 February 2016. <https://wg21.link/p0146r1>
- [P0632r0] Jonathan Müller. Proposal of [[uninitialized]] attribute. 19 January 2017. <https://wg21.link/p0632r0>
- [P2687r0] Bjarne Stroustrup, Gabriel Dos Reis. Design Alternatives for Type-and-Resource Safe C++. 15 October 2022. <https://wg21.link/p2687r0>
- [Randomness] Xi Wang. More randomness or less. 2012-06-25. <https://kqueue.org/blog/2012/06/25/more-randomness-or-less/>
- [Trap] Richard Smith. [NOT FOR REVIEW] Experimental support for zero-or-trap behavior for uninitialized variables. 2020-05-01. <https://reviews.llvm.org/D79249>
- [WinKernel] Joseph Bialek. Join the Windows kernel in wishing farewell to uninitialized plain-old-data structs on the stack. 2018-11-14. <https://twitter.com/JosephBialek/status/1062774315098112001>
- [XNUHeap] Apple Security Engineering and Architecture. Towards the next generation of XNU memory safety: kalloc_type. 2022-10-22. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety-kalloc_type
- [Zeroing] Xi Yang; et al. Why Nothing Matters: The Impact of Zeroing. 2011-10. <https://users.elis.ugent.be/~jsartor/researchDocs/00PSLA2011Zero-submit.pdf>

Chapter 53

wordexpr: compile-time wordle in C++20

👤 Vittorio 📅 2022-2-27 💬 ★★★

It felt wrong to not participate in the Wordle craze, and what better way of doing so than by creating a purely compile-time version of the game in C++20? I proudly present to you... [Wordexpr!](#)

(You can play Wordexpr on Compiler Explorer.)

Carry on reading to understand the magic behind it!

53.1 high-level overview

Wordexpr is played entirely at compile-time as no executable is ever generated - the game is experienced through compiler errors. Therefore, we need to solve a few problems to make everything happen:

1. Produce arbitrary human-readable output as a compiler diagnostic.
2. Random number generation at compile-time.
3. Retain state and keep track of the player's progress in-between compilations.

53.2 error is the new printf

In order to abuse the compiler into outputting errors with an arbitrary string of our own liking, let's start by trying to figure out how to make it print out a simple string literal. The first attempt, `static_assert`, seems promising:

```
1 static_assert(false, "Welcome to Wordexpr!");  
2  
3 error: static assertion failed: Welcome to Wordexpr!  
4     | static_assert(false, "Welcome to Wordexpr!");  
5         ^~~~~
```

However, our delight is short-lived, as `static_assert` only accepts a string literal - a `constexpr` array of characters or `const char*` will not work as an argument:

```

1 constexpr const char* msg = "Welcome to Wordlexpr!";
2 static_assert(false, msg);

1 error: expected string-literal before 'msg'
2 | static_assert(false, msg);
3 |           ^

```

So, how about storing the contents of our string as part of the type of a `struct`, then produce an error containing such type?

```

1 template <char...> struct print;
2 print<'a', 'b', 'c', 'd'> _{};
3
4 error: variable 'print<'a', 'b', 'c', 'd'> _'
5     has initializer but incomplete type
6 3 | print<'a', 'b', 'c', 'd'> _{};
7 |

```

Nice! We are able to see our characters in the compiler output, and we could theoretically mutate or generate the sequence of characters to our liking at compile-time. However, working with a `char...` template parameter pack is very cumbersome, and the final output is not very readable.

C++20's P0732R2: “Class Types in Non-Type Template Parameters” comes to the rescue here! In short, we can use any *literal type* as a non-type template parameter. We can therefore create our own little compile-time string literal type:

```

1 struct ct_str
2 {
3     char       _data[512]{};
4     std::size_t _size{0};
5
6     template <std::size_t N>
7     constexpr ct_str(const char (&str)[N]) : _data{}, _size{N - 1}
8     {
9         for(std::size_t i = 0; i < _size; ++i)
10            _data[i] = str[i];
11     }
12 };

```

We can then accept `ct_str` as a template parameter for `print`, and use the same idea as before:

```

1 template <ct_str> struct print;
2 print<"Welcome to Wordlexpr!"> _{};
3
4 error: variable 'print<ct_str{"Welcome to Wordlexpr!", 21}> _' has
5     initializer but incomplete type
6 22 | print<"Welcome to Wordlexpr!"> _{};
7 |

```

Now we have a way of making the compiler emit whatever we'd like as an error. In fact, we can perform string manipulation at compile-time on `ct_str`:

```

1 constexpr ct_str test()
2 {
3     ct_str s{"Welcome to Wordexpr!"};
4     s._data[0] = 'w';
5     s._data[11] = 'w';
6     s._data[20] = '.';
7     return s;
8 }
9
10 print<test()> _{};
11
12 error: variable 'print<ct_str{"welcome to wordexpr.", 20}> _' has
13     initializer but incomplete type
14 33 | print<test()> _{};
15  |
16      ^

```

By extending `ct_str` with functionalities such as `append`, `contains`, `replace`, etc... we will end up being able to create any sort of string at compile-time and print it out as an error.

First problem solved!

53.3 compile-time random number generation

This is really not a big deal, if we allow our users to provide a seed on the command line via preprocessor defines. Pseudo-random number generation is always deterministic, and the final result only depends on the state of the RNG and the initially provided seed.

```
1 g++ -std=c++20 ./wordexpr.cpp -DSEED=123
```

It is fairly easy to port a common RNG engine such as Mersenne Twister to C++20 `constexpr`. For the purpose of Wordexpr, the modulo operator (`%`) was enough:

```

1 constexpr const ct_str& get_target_word()
2 {
3     return wordlist[SEED % wordlist_size];
4 }
```

Second problem solved!

53.4 retaining state and making progress

If we allow the user to give us a seed via preprocessor defines, why not also allow the user to make progress in the same game session by telling us where they left off last time they played? Think of it as any save file system in a modern game - except that the “save file” is a short string which is going to be passed to the compiler:

```
1 g++ -std=c++20 ./wordleexpr.cpp -DSEED=123 -DSTATE=DJYHULDOPALISHJRBFJNSWAEIM
```

The user doesn't have to come up with the state string themselves - it will be generated by Wordleexpr on every step:

```
1 error: variable 'print<ct_str{"You guessed `crane`. Outcome: `x-xx-`}.
2     You guessed `white`. Outcome: `xxox-`.
3     You guessed `black`. Outcome: `xxxxx`.
4     You guessed `tower`. Outcome: `xxxoo`.
5     To continue the game, pass '-DSTATE=EJYHULDOPALISHJRAVDLYWAEIM'
6     alongside a new guess.", 242}> _' has initializer but incomplete
7     type
8 2612 |         print<make_full_str(SEED, guess, s)> _{};
9     |
```

The state of the game is stored in this simple **struct**:

```
1 struct state
2 {
3     std::size_t _n_guesses{0};
4     ct_str      _guesses[5];
5 };
```

All that's left to do is to define encoding and decoding functions for the state:

```
1 constexpr ct_str encode_state(const state& s);
2 constexpr state decode_state(const ct_str& str);
```

In Wordleexpr, I used a simple Caesar cipher to encode the guesses into the string without making them human-readable. It is not really necessary, but generally speaking another type of compile-time game might want to hide the current state by performing some sort of encoding.

Third problem solved!

53.5 conclusion

I hope you enjoyed this brief explanation of how Wordleexpr works. Remember that you can play it yourself and see the entire source code on Compiler Explorer. Feel free to reach out to ask any question!

Chapter 54

const vs constexpr vs consteval vs constinit in C++20

👤 Bartłomiej Filipek 📅 2022-9-28 🏷★★★

As of C++20, we have four keywords beginning with `const`. What do they all mean? Are they mostly the same? Let's compare them in this article.

54.1 const vs constexpr

`const`, our good old friend from the early days of C++ (and also C), can be applied to objects to indicate immutability. This keyword can also be added to non-static member functions, so those functions can be called on `const` instances of a given type.

`const` doesn't imply any "compile-time" evaluation. The compiler can optimize the code and do so, but in general, `const` objects are initialized at runtime:

```
1 // might be optimized to compile-time if compiled decides...
2 const int importantNum = 42;
3
4 // will be initiatd at runtime
5 std::map<std::string, double> buildMap() { /*...*/ }
6 const std::map<std::string, double> countryToPopulation = buildMap();
```

`const` can sometimes be used in "constant expressions", for example:

```
1 const int count = 3;
2 std::array<double, count> doubles {1.1, 2.2, 3.3};
3
4 // but not double:
5 const double dCount = 3.3;
6 std::array<double, static_cast<int>(dCount)> moreDoubles {1.1, 2.2, 3.3};
7 // error: the value of 'dCount' is not usable in a constant expression
```

See at Compiler Explorer

Let's see the full definition from cppreference:

Defines an expression that can be evaluated at compile time. Such expressions can be used as non-type template arguments, array sizes, and in other contexts that require constant expressions.

If you have a constant integral variable that is `const` initialized, or enumeration value, then it can be used at constant expression.

Since C++11, we have a new keyword - `constexpr` - which pushed further the control over variables and functions that can be used in constant expressions. Now it's not a C++ trick or a special case, but a complete, easier-to-understand solution.

```
// fine now:
constexpr double dCount = 3.3;
std::array<double, static_cast<int>(dCount)> doubles2 {1.1, 2.2, 3.3};
```

Above, the code uses `double`, which is allowed in constant expressions.

We can also create and use simple structures:

```
1 #include <array>
2 #include <cstdlib>
3
4 struct Point {
5     int x { 0 };
6     int y { 0 };
7
8     constexpr int distX(const Point& other) const { return abs(x - other.x); }
9 };
10
11 int main() {
12     constexpr Point a { 0, 1 };
13     constexpr Point b { 10, 11 };
14     static_assert(a.distX(b) == 10);
15
16     // but also at runtime:
17     Point c { 100, 1 };
18     Point d { 10, 11 };
19     return c.distX(d);
20 }
```

Run @Compiler Explorer

In summary:

- `const` can be applied to all kinds of objects to indicate their immutability
- `const` integral type with constant initialization can be used in constant expression

- `constexpr` object, by definition, can be used in constant expressions
- `constexpr` can be applied to functions to show that they can be called to produce constant expressions (they can also be called at runtime)
- `const` can be applied to member functions to indicate that a function doesn't change data members (unless mutable),

54.2 constexpr vs consteval

Fast forward to C++20, we have another keyword: `consteval`. This time it can be applied only to functions and forces all calls to happen at compile time.

For example:

```

1  consteval int sum(int a, int b) {
2      return a + b;
3  }
4
5  constexpr int sum_c(int a, int b) {
6      return a + b;
7  }
8
9  int main() {
10     constexpr auto c = sum(100, 100);
11     static_assert(c == 200);
12
13     constexpr auto val = 10;
14     static_assert(sum(val, val) == 2*val);
15
16     int a = 10;
17     int b = sum_c(a, 10); // fine with constexpr function
18
19     // int d = sum(a, 10); // error! the value of 'a' is
20                 // not usable in a constant expression
21 }
```

See [@Compiler Explorer](#).

Immediate functions can be seen as an alternative to function-style macros. They might not be visible in the debugger (inlined)

Additionally, while we can declare a `constexpr` variable, there's no option to declare a `consteval` variable.

```
// consteval int some_important_constant = 42; // error
```

In summary:

- `consteval` can only be applied to functions

- `constexpr` can be applied to functions and also variables
- `consteval` forces compile time function evaluation; the `constexpr` function can be executed at compile time but also at runtime (as a regular function).

54.3 `constexpr` vs `constinit`

And now the last keyword: `constinit`.

`constinit` forces constant initialization of static or thread-local variables. It can help to limit static order initialization fiasco by using precompiled values and well-defined order rather than dynamic initialization and linking order...

```

1 #include <array>
2
3 // init at compile time
4 constexpr int compute(int v) { return v*v*v; }
5 constinit int global = compute(10);
6
7 // won't work:
8 // constinit int another = global;
9
10 int main() {
11     // but allow to change later...
12     global = 100;
13
14     // global is not constant!
15     // std::array<int, global> arr;
16 }
```

See at Compiler Explorer

Contrary to `const` or `constexpr`, it doesn't mean that the object is immutable. What's more `constinit` variable cannot be used in constant expressions! That's why you cannot init another with `global` or use `global` as an array size.

In summary:

- `constexpr` variables are constant and usable in constant expressions
- `constinit` variables are not constant and cannot be used in constant expressions
- `constexpr` can be applied on local automatic variables; this is not possible with `constinit`, which can only work on static or `thread_local` objects
- you can have a `constexpr` function, but it's not possible to have a `constinit` function declaration

54.4 Mixing

Can we have mixes of those keywords?

```

1 // possible and compiles... but why not use constexpr?
2 constinit const int i = 0;
3
4 // doesn't compile:
5 constexpr constinit int i = 0;
6
7 // compiles:
8 const constexpr int i = 0;
```

And we can have `const` and `constexpr` member functions:

```

1 struct Point {
2     int x { 0 };
3     int y { 0 };
4
5     constexpr int distX(const Point& other) const { return abs(x - other.x); }
6     constexpr void mul(int v) { x *= v; y *= v; }
7 };
```

In the above scenario, `constexpr` means that the function can be evaluated for constant expressions, but `const` implies that the function won't change its data members. There's no issue in having only a `constexpr` function like `mul`:

```

1 constexpr int test(int start) {
2     Point p { start, start };
3     p.mul(10); // changes p!
4     return p.x;
5 }
6
7 int main() {
8     static_assert(test(1) == 10);
9 }
```

Run at Compiler Explorer

54.5 Summary

After seeing some examples, we can build the following summary table:

keyword	on auto variables	to static/thread_local variables	to functions	constant expressions
<code>const</code>	yes	yes	as <code>const</code> member functions	sometimes
<code>constexpr</code>	yes or implicit (in <code>constexpr</code> functions)	yes	to indicate <code>constexpr</code> functions	yes
<code>consteval</code>	no	no	to indicate <code>constexpr</code> functions	yes (as a result of a function call)
<code>constinit</code>	no	to force constant initialization	no	no, a <code>constinit</code> variable is not a <code>constexpr</code> variable

You can also find another cool article from Marius Bancila where he also compared those keywords (back in 2019): Let there be constants!.

As of C++23 there's no new `const` keyword added, so the list I showed in this article should be valid for a couple of years at least :)

Part VII

Algorithms

本 Part 主要集中于 Modern C++ 中关于算法的新特性，内容不多，包含 Ranges Algorithms 和 C++23 的 Fold Algorithms，还有一篇关于整型格式化的算法，是一篇五星文章。

这部分的主要目标是构建对于新算法组件的综合性认识，而非真得像算法书那样去讲解数学原理，除了最后一篇，难度不大。

2023 年 5 月 13 日
里缪

Chapter 55

C++20 Ranges Algorithms

👤 Bartłomiej Filipek 📅 2022-04-25 / 2022-05-16 / 2022-06-13 🎨 ★★★

55.1 7 Non-modifying Operations

C++20’s Ranges offer alternatives for most of `<algorithm>`’s¹. This time I’d like to show you ten non-modifying operations. We’ll compare them with the “old” standard version and see their benefits and limitations.

Let’s go.

55.1.1 Before we start

Key observations for `std::ranges` algorithms:

- Ranges algorithms are defined in the `<algorithm>` header, while the ranges infrastructure and core types are defined in the `<ranges>` header.
- Usually, there are at least two overloads for range algorithms: with a pair of iterators and an overload with a single range argument.
- The version that returns a subrange or an iterator and takes a range returns a borrowed range or a borrowed iterator. This helps detect iterators to temporary ranges.
- The range versions take “projections,” which sometimes allows more flexibility; for example, you can sort against some selected members or perform additional transformations before the comparison.
 - See my separate article on this powerful feature: C++20 Ranges, Projections, `std::invoke` and `if constexpr` - C++ Stories¹
- The ranges version doesn’t have a parallel execution option (you cannot pass the `std::execution` policy).
- The ranges algorithms, similarly to standard algorithms as of C++20, are also `constexpr`.
- As of C++20, there are no ranges numerical algorithms corresponding to the `<numeric>` header.

¹<https://www.cppstories.com/2020/10/understanding-invoke.html/>

Below, you can find examples showing a standard algorithm and an alternative version with ranges. They illustrate some basic concepts and try not to use advanced ranges composition or views. We'll go with the order found at [cppreference/algorithms](#), and in this part, we'll cover “Non-modifying sequence operations.”

55.1.2 all_of, any_of, none_of

A standard algorithm:

```

1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4 #include <ranges>
5
6 int main() {
7     const std::vector nums = {1, 2, 3, -4, 5, 6, 7, 8};
8
9     auto is_positive = [] (const auto& v) { return v > 0; };
10
11    // standard version:
12    auto res = std::all_of(begin(nums), end(nums), is_positive);
13    std::cout << "std::all_of: " << res << '\n';
14
15    res = std::any_of(begin(nums), end(nums), is_positive);
16    std::cout << "std::any_of: " << res << '\n';
17 }
```

And the ranges version:

```

1 // https://godbolt.org/z/r5M7vbee4
2 // ranges version:
3 res = std::ranges::all_of(nums, is_positive);
4 std::cout << "std::ranges::all_of: " << res << '\n';
5
6 res = std::ranges::any_of(nums, is_positive);
7 std::cout << "std::ranges::any_of: " << res << '\n';
```

We can also write a more complex example where scan a container of custom types:

```

1 // https://godbolt.org/z/scon89cc5
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6
7 struct Product {
```

```

8     std::string name_;
9     double value_ { 0.0 };
10    };
11
12 int main() {
13     const std::vector<Product> prods {
14         { "box", 10.0 }, {"tv", 100.0}, {"none", -1.0}
15     };
16
17     auto is_positive = [](const auto& v) { return v > 0; };
18     auto is_positive_val = [](const Product& p) {
19         return p.value_ > 0;
20     };
21
22     // standard version:
23     auto res = std::all_of(begin(prods), end(prods), is_positive_val);
24     std::cout << "std::all_of: " << res << '\n';
25
26     res = std::any_of(begin(prods), end(prods), is_positive_val);
27     std::cout << "std::any_of: " << res << '\n';
28
29     // ranges version:
30     res = std::ranges::all_of(prods, is_positive, &Product::value_);
31     std::cout << "std::ranges::all_of: " << res << '\n';
32
33     res = std::ranges::any_of(prods, is_positive, &Product::value_);
34     std::cout << "std::ranges::any_of: " << res << '\n';
35 }
```

In the ranges version, we can still use `is_positive`, a generic predicate, but I used a projection that only “takes” `Product::value_` and passes it into the predicate. In the standard case, I had to write a custom lambda aware of the `Product` type.

55.1.3 for_each

An alternative to a good range based for loop:

```

1 // https://godbolt.org/z/fvfrdEqeb
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6
7 struct Product {
```

```

8     std::string name_;
9     double value_ { 0.0 };
10    };
11
12 int main() {
13     const std::vector<Product> prods {
14         { "box", 10.0 }, {"tv", 100.0}, {"none", -1.0}
15     };
16
17     auto out = [](<const auto& v) { std::cout << v << ", "; };
18
19     // standard version:
20     std::cout << "std::for_each: \n";
21     std::for_each(begin(prods), end(prods), [](<const Product& p){
22         std::cout << p.name_ << ", " << p.value_ << '\n';
23     });
24
25     std::cout << "std::for_each only names reverse: \n";
26     std::for_each(rbegin(prods), rend(prods), [](<const Product& p){
27         std::cout << p.name_ << '\n';
28     });
29
30     // ranges version:
31     std::cout << "std::ranges::for_each: \n";
32     std::ranges::for_each(prods, [<const Product& p) {
33         std::cout << p.name_ << ", " << p.value_ << '\n';
34     });
35
36     std::cout << "std::ranges::for_each only names in reverse: \n";
37     std::ranges::for_each(prods | std::views::reverse,
38                           out, &Product::name_);
39 }

```

The exciting part is that printing in reverse order in the standard version requires to use `rbegin/rend` iterators and then a custom unary function to print the exact data member from the `Product` class. While with ranges we can apply `views::reverse`, use a simple output function, and then a projection.

What's missing is the parallel algorithm version of the ranges algorithms:

```

1 // standard:
2 std::for_each(std::execution::par, begin(prods), end(prods), /*...*/);
3 // no ranges version...
4 // std::ranges::for_each(std::execution::par, prods, /*... */); // doesn't compile...

```

Parallel versions are lacking for **all** ranges algorithms, not just for `for_each`.

55.1.4 count_if

In the example below we'll count Products that have name starting with "no" :

```

1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4 #include <ranges>
5
6 struct Product {
7     std::string name_;
8     double value_ { 0.0 };
9 };
10
11 int main() {
12     const std::vector<Product> prods {
13         { "box", 10.0 }, {"tv", 100.0}, {"none", -1.0},
14         { "car", 1000.0 }, {"toy", 40.0}, {"none", 0.0}
15     };
16
17     // standard version:
18     auto res = std::count_if(begin(prods), end(prods), [] (const Product& p) {
19         return p.name_.starts_with("no");
20     });
21     std::cout << "std::count_if: " << res << '\n';
22
23     // ranges version:
24     res = std::ranges::count_if(prods, [] (const Product& p) {
25         return p.name_.starts_with("no");
26     });
27     std::cout << "std::ranges::count_if: " << res << '\n';
28
29     // alternative version for "none":
30     res = std::ranges::count(prods, std::string{"none"}, &Product::name_);
31     std::cout << "std::ranges::count: " << res << '\n';
32 }
```

The example shows three approaches, and the last one uses a projection to check only the `Product::name_` data member. In that approach, we search exactly for "none" so it's stricter than with `starts_with`.

55.1.5 find_if

So far, our text algorithms have returned boolean or integral values, but with `find*` functions, we have iterators (or subranges) that show the same occurrence.

See the example:

```

1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4 #include <ranges>
5
6 struct Product {
7     std::string name_;
8     double value_ { 0.0 };
9 };
10
11 int main() {
12     const std::vector<Product> prods {
13         { "box", 10.0 }, {"tv", 100.0}, {"rocket", 10000.0},
14         { "car", 1000.0 }, {"toy", 40.0}, {"none", 0.0}
15     };
16
17     // standard version:
18     auto it = std::find_if(begin(prods), end(prods), [] (const Product& p) {
19         return p.name_.starts_with("ro");
20     });
21     if (it != end(prods))
22         std::cout << "std::find_if: " << it->name_ << '\n';
23
24     // ranges version:
25     auto res = std::ranges::find_if(prods, [] (const Product& p) {
26         return p.name_.starts_with("ro");
27     });
28     if (res != end(prods))
29         std::cout << "std::ranges::find_if: " << res->name_ << '\n';
30 }

```

Like with many other algorithms, there's also a "regular" version where you can pass two iterators:

```

1 it = std::ranges::find_if(begin(prods), end(prods), [] (const Product& p) {
2     return p.name_.starts_with("ro");
3 });

```

The version that takes a single range is special, as it returns a **borrowed** iterators. This special type allows checking for temporary/lifetime object issues. This is not possible when you pass two iterators (because the container is present somewhere), but possible with a single temporary range:

```

1 struct Product {
2     std::string name_;
3     double value_ { 0.0 };
4 };

```

```

5
6 std::vector<Product> GetProds() {
7     return {
8         { "box", 10.0 }, {"tv", 100.0}, {"rocket", 10000.0},
9         { "car", 1000.0 }, {"toy", 40.0}, {"none", 0.0}
10    };
11 }
12
13 int main() {
14     auto it = std::ranges::find_if(GetProds(), [](const Product& p) {
15         return p.name_.starts_with("ro");
16     });
17     std::cout << "std::ranges::find_if: " << it->name_ << '\n';
18 }
```

This doesn't compile and you'll see the following error:

```
error: base operand of '->' has non-pointer type std::ranges::dangling
22 |     std::cout << "std::ranges::find_if: " << it->name_ << '\n';
|                                     ^~
```

As you can see, the compiler checked that `GetProds()` returns a temporary, and the iterator that we'd find would be dangling. See the code: <https://godbolt.org/z/f5v1P4rGo>.

55.1.6 `find_first_of`

Let's have a look at another `find*` function alternative that searches for multiple elements at once.

```

1 // https://godbolt.org/z/PW5G8xY37
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6
7 struct Product {
8     std::string name_;
9     double value_ { 0.0 };
10
11     friend bool operator==(const Product& a, const Product& b) {
12         return a.name_ == b.name_ && abs(a.value_ - b.value_) < 0.0001;
13     }
14 };
15
16 int main() {
17     const std::vector<Product> prods {
18         { "box", 10.0 }, {"default", 0.0 }, {"tv", 100.0}, {"rocket", 10000.0},
```

```

19         { "car", 1000.0 }, {"toy", 40.0}, {"none", 0.0 }, { "ball", 40.0 }
20     };
21
22     const std::vector<Product> invalids {
23         {"default", 0.0 }, {"none", 0.0 }
24     };
25
26     // standard version:
27     auto it = std::find_first_of(begin(prods), end(prods), begin(invalids), end(invalids));
28     if (it != end(prods)) {
29         std::cout << "std::find_first_of: " << it->name_ << " at: "
30                         << std::distance(begin(prods), it) << '\n';
31         auto it2 = std::find_first_of(std::next(it),
32                                     end(prods), begin(invalids), end(invalids));
33         if (it2 != end(prods))
34             std::cout << "std::find_first_of: " << it2->name_ << " at: "
35                         << std::distance(begin(prods), it2) << '\n';
36     }
37
38     // ranges version:
39     const std::array<std::string, 2> arrInvalids{"default", "none"};
40     auto res = std::ranges::find_first_of(prods, arrInvalids,
41                                         std::ranges::equal_to{}, &Product::name_);
42     if (res != end(prods)) {
43         const auto pos = std::distance(begin(prods), res);
44         std::cout << "std::ranges::find_first_of: " << res->name_
45                         << " at: " << pos << '\n';
46
47         auto res2 = std::ranges::find_first_of(prods | std::views::drop(pos+1), arrInvalids,
48                                         std::ranges::equal_to{}, &Product::name_);
49         if (res2 != end(prods)) {
50             std::cout << "std::ranges::find_first_of: " << res2->name_
51                         << " at: " << std::distance(begin(prods), res2) << '\n';
52         }
53     }
54 }
```

`std::find_first_of` takes two pairs of iterators. I wanted to find “invalid” products in my prod sequence in the example. Since I’m comparing products, I had to define `operator==` for my structure. Alternatively, I can provide a binary operation and then compare just the names:

```

1 auto cmpNames = [](const Product& a, const Product& b) {
2     return a.name_ == b.name_;
3 };
```

```

4
5 auto it = std::find_first_of(begin(prods), end(prods),
6                               begin(invalids), end(invalids), cmpNames);
7 if (it != end(prods)) {
8     // ...
9 }
```

In the ranges version I can use projections and default comparator to achieve similar effect:

```

1 const std::array<std::string, 2> arrInvalids{"default", "none"};
2 auto res = std::ranges::find_first_of(prods, arrInvalids,
3                                     std::ranges::equal_to{}, &Product::name_);
```

The interesting part later is that for the second search I can use drop to skip first N elements from the range:

```

1 auto res2 = std::ranges::find_first_of(prods | std::views::drop(pos+1),
2 arrInvalids, std::ranges::equal_to{}, &Product::name_);
```

Alternatively you can also use a version with two pairs of iterators:

```

1 auto res2 = std::ranges::find_first_of(std::next(res), end(prods),
2 begin(arrInvalids), end(arrInvalids),
3 std::ranges::equal_to{}, &Product::name_);
```

55.1.7 mismatch

With the `mismatch` algorithm we can find the first place where two ranges differ:

```

1 // https://godbolt.org/z/zGh3cj9e5
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6 #include <iomanip> // quoted
7
8 int main() {
9     const std::string firstStr = "Hello Super World";
10    const std::string secondStr = "Hello Amazing World";
11
12    std::cout << "mismatch for " << std::quoted(firstStr)
13        << " and " << std::quoted(secondStr) << '\n';
14
15    // standard version:
16    auto [first, second] = std::mismatch(begin(firstStr), end(firstStr), begin(secondStr));
17    {
18        const auto pos = std::distance(begin(firstStr), first);
```

```

19         std::cout << "std::mismatch: at pos " << pos << '\n';
20     }
21
22     // ranges version:
23     auto res = std::ranges::mismatch(firstStr, secondStr);
24     {
25         const auto pos = std::distance(begin(firstStr), res.in1);
26         std::cout << "std::ranges::mismatch: at pos " << pos << '\n';
27     }
28 }
```

The ranges version returns:

```

1 template<class I1, class I2>
2 using mismatch_result = ranges::in_in_result<I1, I2>;
```

Which is a pair of two iterators, but we can access them via `.in1` and `.in2`.

Why not a simple range? At cpp reference² we can see the following sentence:

Unlike `std::pair` and `std::tuple`, this class template has data members of meaningful names.

The result works fine with structured binding, so you can write:

```

1 auto [n1, n2] = std::ranges::mismatch(firstStr, secondStr);
2 const auto pos = std::distance(begin(firstStr), n1);
3 std::cout << "std::ranges::mismatch: at pos " << pos << '\n';
```

The code is almost the same as the standard version.

55.1.8 search

Searching for patterns in the other range/container:

```

1 // https://godbolt.org/z/T191KfrKj
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6 #include <functional> // searchers
7 #include <iomanip>
8
9 int main() {
10     const std::string testString = "Hello Super World";
11     const std::string needle = "Super";
12
13     std::cout << "looking for " << std::quoted(needle)
```

²https://en.cppreference.com/w/cpp/algorithm/ranges/return_types/in_in_result

```

14         << " in " << std::quoted(testString) << '\n';
15
16     // standard version:
17     auto it = std::search(testString.begin(), testString.end(),
18                           std::boyer_moore_searcher(needle.begin(), needle.end()));
19
20     if (it != testString.end()) {
21         const auto pos = std::distance(testString.begin(), it);
22         std::cout << "std::search: found at pos " << pos << '\n';
23     }
24
25     // ranges version:
26     auto res = std::ranges::search(testString, needle);
27     if (!res.empty()) {
28         const auto first = std::distance(testString.begin(), res.begin());
29         const auto last = std::distance(testString.begin(), res.end());
30         std::cout << "std::ranges::search: found between "
31                     << first << " and " << last << '\n';
32     }
33 }
```

The standard version returns an iterator to the first string where the second string starts (or end() if not there). While the ranges version returns a subrange (or a borrowed_subrange).

We can also use projections for checking in a case insensitive way:

```

1 // https://godbolt.org/z/1WTa655d6
2 // ranges version:
3 const std::string testString2 = "hello abc world";
4 const std::string needle2 = "ABC";
5 std::cout << "looking for " << std::quoted(needle2) << " in "
6             << std::quoted(testString2) << '\n';
7
8 res = std::ranges::search(testString2, needle2,
9                           std::ranges::equal_to{}, ::toupper, ::toupper);
10 if (!res.empty())
11 {
12     const auto first = std::distance(testString2.begin(), res.begin());
13     const auto last = std::distance(testString2.begin(), res.end());
14     std::cout << "std::ranges::search: found between "
15                     << first << " and " << last << '\n';
16 }
```

You can read more about searches in my two articles:

- Speeding up Pattern Searches with Boyer-Moore Algorithm from C++17 - C++ Stories³
- Preprocessing Phase for C++17's Searchers - C++ Stories

The other function `ranges::search_n` is handy for finding N occurrences of a given value in the input range:

```

1 // https://godbolt.org/z/1vnGa4nhj
2 #include <algorithm>
3 #include <iostream>
4 #include <ranges>
5 #include <iomanip>
6
7 int main() {
8     const std::string sequence = "CTGCCAGGGTTT";
9     const char letter = 'C';
10    const size_t count = 3;
11
12    std::cout << "looking for " << count << " "
13                  << letter << "'s in " << std::quoted(sequence) << '\n';
14
15    // standard version:
16    auto it = std::search_n(begin(sequence), end(sequence), count, letter);
17
18    if (it != end(sequence))
19    {
20        const auto pos = std::distance(begin(sequence), it);
21        std::cout << "std::search_n: found at pos " << pos << '\n';
22    }
23
24    // ranges version:
25    auto res = std::ranges::search_n(sequence, count, letter);
26    if (!res.empty())
27    {
28        const auto first = std::distance(begin(sequence), res.begin());
29        const auto last = std::distance(begin(sequence), res.end());
30        std::cout << "std::ranges::search_n: found between "
31                  << first << " and " << last << '\n';
32    }
33 }
```

In the standard version, there are no special searchers; you can only invoke it using parallel algorithms.

³<https://www.cppstories.com/2018/08/searchers/>

55.1.9 Summary

In this section, we covered seven different algorithm “types” in the category of non-modifying operations: checking some predicate on all/none/some elements, searching, finding, general iteration. In total, there were more than 10 different examples.

The ranges algorithms offer an easier way to pass the “whole” container - just one argument, rather than to iterators. They also allow for projections and have a way to detect iterators to a temporary range. They also have limitations, like the lack of advanced searchers or parallel execution mode.

Stay tuned for the next section, where we’ll discuss remaining operations like `std::transform`, sorting, min/max, partitioning, numerics, and we’ll see what we’ll get soon in C++23.

55.2 11 Modifying Operations

In the previous section in the Ranges series, I covered some basics and non-modifying operations. Today it’s time for algorithms like `transform`, `copy`, `generate`, `shuffle`, and many more.... and there’s `rotate` as well :)

This section will cover some of the algorithms that allow changing the sequence, like copying, removing, transforming, or generating elements.

Let’s go.

55.2.1 `copy_if`

There are many variations of this core algorithm: `copy`, `copy_if`, `copy_n` or even `copy_backward`.

In a basic form `copy_if` is defined as follows:

```
1 // skipping all concept/templates declaration
2 constexpr copy_if_result<ranges::borrowed_iterator_t<R>, 0>
3     copy_if( R&& r, 0 result, Pred pred, Proj proj = {} );
```

Let’s try a basic example with:

```
1 // https://godbolt.org/z/8ovTxrer6
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6
7 struct Product {
8     std::string name_;
9     double value_ { 0.0 };
10};
11
12 std::ostream& operator<<(std::ostream& os, const Product& p) {
13     os << p.name_ << ", " << p.value_;
14     return os;
```

```

15 }
16
17 int main() {
18     const std::vector<Product> prods {
19         { "box", 10.0 }, {"tv", 100.0}, {"rocket", 10000.0},
20         { "car", 1000.0 }, {"toy", 40.0}, {"none", 0.0}
21     };
22
23     // standard version:
24     std::copy_if(begin(prods), end(prods),
25                  std::ostream_iterator<Product>(std::cout, "; "),
26                  [] (const Product& p){
27                     return !p.name_.starts_with("none");
28                 });
29     std::cout << '\n';
30
31     // ranges version:
32     std::ranges::copy_if(prods,
33                         std::ostream_iterator<Product>(std::cout, "; "),
34                         [] (const Product& p){
35                             return !p.name_.starts_with("none");
36                         });
37 }
```

In the example, I copy elements from the vector into the output stream. Additionally, as a filter step, I want only products that are not “none”. Since we copy whole elements to the stream, I had to implement `operator<<` for the `Product` class.

Thanks to projections, I could also write a following version:

```

1 std::ranges::copy_if(prods,
2                     std::ostream_iterator<Product>(std::cout, "; "),
3                     [] (const std::string& name){
4                         return !name.starts_with("none");
5                     },
6                     &Product::name_);
```

The code is a bit longer, but now the predicate takes a string rather than a whole `Product` object.

See more at `ranges::copy`, `ranges::copy_if` (<https://en.cppreference.com/w/cpp/algorithm/ranges/copy>).

55.2.2 fill

```

1 // https://godbolt.org/z/vTcb4EGc7
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
```

```

5 #include <ranges>
6
7 struct Product {
8     std::string name_;
9     double value_ { 0.0 };
10
11     Product& operator=(int i) { name_ += std::to_string(i); return *this; }
12 };
13
14 std::ostream& operator<<(std::ostream& os, const Product& p) {
15     os << p.name_ << ", " << p.value_;
16     return os;
17 }
18
19 int main() {
20     std::vector<Product> prods{7, {"Box ", 1.0}};
21
22     // standard version:
23     std::fill(begin(prods), end(prods), 4);
24     std::ranges::copy(prods, std::ostream_iterator<Product>(std::cout, "; "));
25     std::cout << '\n';
26
27     // ranges version:
28     std::ranges::fill(prods, 2);
29     std::ranges::copy(prods, std::ostream_iterator<Product>(std::cout, "; "));
30 }

```

The `fill` algorithm walks on the range and then performs the assignment with the `value` you pass. The `value` might have been of a different type than the elements in the container.

```

1 while (first != last)
2     *first++ = value;

```

In the example, I used a class with a custom conversion operator, and that's why we can use it to modify the `name_` data member based on the integral input value.

See more at `ranges::fill` (<https://en.cppreference.com/w/cpp/algorithm/ranges/fill>).

55.2.3 generate

While `fill()` uses the same value to assign to all elements, `generate()` uses a function object to generate the value. In the example we can simulate the `iota` generation:

```

1 // https://godbolt.org/z/anT3zrjYe
2 #include <algorithm>
3 #include <vector>

```

```

4  #include <iostream>
5  #include <ranges>
6
7  struct Product {
8      std::string name_;
9      double value_ { 0.0 };
10
11     Product& operator=(int i) { name_ += std::to_string(i); return *this; }
12 };
13
14 std::ostream& operator<<(std::ostream& os, const Product& p) {
15     os << p.name_ << ", " << p.value_;
16     return os;
17 }
18
19 int main() {
20     std::vector<Product> prods{7, {"Box ", 1.0}};
21
22     // standard version:
23     std::generate(begin(prods), end(prods), [v = 0] () mutable {
24         return v++;
25     });
26     std::ranges::copy(prods, std::ostream_iterator<Product>(std::cout, "; "));
27     std::cout << '\n';
28
29     // ranges version:
30     std::ranges::generate(prods, [v = 0] () mutable {
31         return ++v;
32     });
33     std::ranges::copy(prods, std::ostream_iterator<Product>(std::cout, "; "));
34 }
```

The output:

```
Box 0, 1; Box 1, 1; Box 2, 1; Box 3, 1; Box 4, 1; Box 5, 1; Box 6, 1;
Box 01, 1; Box 12, 1; Box 23, 1; Box 34, 1; Box 45, 1; Box 56, 1; Box 67, 1;
```

See more at `ranges::generate` (<https://en.cppreference.com/w/cpp/algorithm/ranges/generate>). And there's also an alternative version with `_n`: `ranges::generate_n`.

55.2.4 transform

`transform()` is a robust algorithm that has many variations.

In a basic form it looks as follows:

```
transform( R&& r, O result, F op, Proj proj = {} );
```

It takes a range `r` and then uses `op` to transform elements from that range and output it into the `result`, which is an iterator.

See the basic example:

```

1 // https://godbolt.org/z/89qMoYdb5
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6
7 struct Product {
8     std::string name_;
9     double value_ { 0.0 };
10 };
11
12 int main() {
13     std::vector<Product> prods{7, {"Box ", 1.0}};
14
15     // standard version:
16     std::transform(begin(prods), end(prods), begin(prods),
17                   [v = 0](const Product &p) mutable {
18                     return Product { p.name_ + std::to_string(v++), 1.0};
19                 });
20     for (auto &p : prods) std::cout << p.name_ << ", ";
21     std::cout << '\n';
22
23     // ranges version:
24     std::ranges::transform(prods, begin(prods), [v = 0](const std::string &n) mutable {
25         return Product { n + std::to_string(v++), 1.0};
26     }, &Product::name_);
27     for (auto &p : prods) std::cout << p.name_ << ", ";
28 }
```

The output:

```
Box 0, Box 1, Box 2, Box 3, Box 4, Box 5, Box 6,
Box 00, Box 11, Box 22, Box 33, Box 44, Box 55, Box 66,
```

The example transforms the same container but adds numbers - generated through a function - to each name.

There's also a version that takes two ranges and combines them with a binary operation:

```
transform( R1&& r1, R2&& r2, O result, F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {} );
```

We can use this version to “join” two containers and produce a single value:

```

1 // https://godbolt.org/z/Wo1WzochE
2 std::vector<Product> prods{7, {"Box ", 1.0}};
3 std::vector<int> numbers{1, 2, 3, 4, 5, 6, 7};
4
5 std::ranges::transform(prods, numbers, begin(prods),
6 [] (const Product& p, int v) {
7     return Product { p.name_ + std::to_string(v), 1.0};
8 });
9 for (auto &p : prods) std::cout << p.name_ << ", ";

```

See more at `ranges::transform` (<https://en.cppreference.com/w/cpp/algorithm/ranges/transform>).

55.2.5 remove

In C++20, we have a more efficient way to remove and erase elements from various containers. See `std::erase_if`, a set of overloaded functions for consistent container erasure. You can read more in my article: 20 Smaller yet Handy C++20 Features - Consistent Container Erasure⁴.

For completeness, let's compare all three versions:

```

1 // https://godbolt.org/z/4cYjP83cP
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6
7 struct Product {
8     std::string name_;
9     double value_ { 0.0 };
10 };
11
12 int main() {
13     const std::vector<Product> prods {
14         { "box", 10.0 }, {"tv", 100.0}, {"rocket", 10000.0},
15         {"no prod", 0.0}, { "car", 1000.0 }, {"toy", 40.0}, {"none", 0.0}
16     };
17
18     auto printCont = [] (const std::vector<Product>& cont) {
19         for (auto &p : cont) std::cout << p.name_ << ", ";
20         std::cout << '\n';
21     };
22     std::cout << "removing products starting with \"no\"\n";
23     printCont(prods);
24

```

⁴<https://www.cppstories.com/2022/20-smaller-cpp20-features/#15-consistent-container-erasure>

```

25     auto checkNoPrefix = [&](const Product& p) { return p.name_.starts_with("no"); };
26
27     // standard version:
28     auto tempProds = prods;
29     tempProds.erase(std::remove_if(tempProds.begin(), tempProds.end(),
30         checkNoPrefix), tempProds.end());
31     printCont(tempProds);
32
33     // ranges version:
34     tempProds = prods;
35     tempProds.erase(std::ranges::remove_if(
36         tempProds, checkNoPrefix).begin(), tempProds.end());
37     printCont(tempProds);
38
39     // C++20 version:
40     tempProds = prods;
41     std::erase_if(tempProds, checkNoPrefix);
42     printCont(tempProds);
43 }
```

The ranges version can shorten the call to:

```
tempProds.erase(std::remove_if(tempProds.begin(), tempProds.end(),
    checkNoPrefix), tempProds.end());
```

into:

```
tempProds.erase(std::ranges::remove_if(tempProds, checkNoPrefix).begin(), tempProds.end());
```

But, in my opinion, this doesn't look that much better. `ranges::remove_if` returns a subrange, so you need to get its `begin()` and possibly `end()` anyway.

It's much easier to write:

```
std::erase_if(tempProds, checkNoPrefix);
```

See more at `ranges::remove`/`ranges::remove_if` (<https://en.cppreference.com/w/cpp/algorithm/ranges/remove>) and also `std::erase`, `std::erase_if` (`std::vector`) (<https://en.cppreference.com/w/cpp/container/vector/erase2>) (each container has its own overload for `std::erase`).

55.2.6 replace

How to replace elements inside a container:

```

1 // https://godbolt.org/z/vPP4dbGP3
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
```

```

6
7 struct Product {
8     std::string name_;
9     double value_ { 0.0 };
10
11    friend bool operator==(const Product& a, const Product& b) {
12        return a.name_ == b.name_ && abs(a.value_ - b.value_) < 0.0001;
13    }
14};
15
16 std::ostream& operator<<(std::ostream& os, const Product& p) {
17     os << p.name_ << ", " << p.value_;
18     return os;
19 }
20
21 int main() {
22     std::vector<Product> prods {
23         { "box", 10.0 }, {"tv", 100.0}, {"rocket", 10000.0},
24         { "car", 1000.0 }, {"toy", 40.0}, {"none", 0.0},
25         {"invalid", 0.0}, { "invalid", -10.0 }
26     };
27
28     std::ostream_iterator<Product> out_iter(std::cout, "; ");
29
30     // standard version:
31     std::cout << "before: \n";
32     std::copy(begin(prods), end(prods), out_iter);
33     std::replace(begin(prods), end(prods), Product{"none", 0.0}, Product{"default", 10.0});
34     std::cout << "\nafter: \n";
35     std::copy(begin(prods), end(prods), out_iter);
36     std::cout << '\n';
37
38     // ranges version:
39     std::cout << "before: \n";
40     std::ranges::copy(prods, out_iter);
41     std::ranges::replace(prods, "invalid", Product{"default", 10.0}, &Product::name_);
42     std::cout << "\nafter: \n";
43     std::ranges::copy(prods, out_iter);
44     std::cout << '\n';
45 }

```

The output:

before:

```

box, 10; tv, 100; rocket, 10000; car, 1000; toy, 40; none, 0; invalid, 0; invalid, -10;
after:
box, 10; tv, 100; rocket, 10000; car, 1000; toy, 40; default, 10; invalid, 0; invalid, -10;
before:
box, 10; tv, 100; rocket, 10000; car, 1000; toy, 40; default, 10; invalid, 0; invalid, -10;
after:
box, 10; tv, 100; rocket, 10000; car, 1000; toy, 40; default, 10; default, 10; default, 10;

```

The interesting part is that in the standard version we compare a value against objects stored in the container:

```

1 for (; first != last; ++first) {
2     if (*first == old_value) {
3         *first = new_value;
4     }
5 }

```

And that's why I had to define a comparison operator == (or a spaceship <=> to be more flexible).

In the ranges version we can use projection as the comparison is a bit different:

```

1 for (; first != last; ++first) {
2     if (old_value == std::invoke(proj, *first)) {
3         *first = new_value;
4     }
5 }

```

And in the example, there's no need to have the == operator, as we can compare strings directly. This gives us more flexibility, as we can find more "Invalid" values (the value of value_ is not checked now to catch both - 0.0 and -10.0 and fix them).

See more `ranges::replaceranges::replace_if` (<https://en.cppreference.com/w/cpp/algorithm/ranges/replace>).

55.2.7 reverse

Let's try the version with a reverse copy that outputs to the stream:

```

1 // https://godbolt.org/z/14E5e6KGK
2 #include <algorithm>
3 #include <vector>
4 #include <iostream>
5 #include <ranges>
6
7 int main() {
8     const std::vector numbers {
9         "one", "two", "three", "four", "five", "six"
10    };
11
12    auto outStream = std::ostream_iterator<std::string>(std::cout, "; ");

```

```

13
14    // standard version:
15    std::copy(begin(numbers), end(numbers), outStream);
16    std::cout << '\n';
17    std::reverse_copy(begin(numbers), end(numbers), outStream);
18
19    // ranges version:
20    std::cout << "\nRanges\n";
21    std::ranges::copy(numbers, outStream);
22    std::cout << '\n';
23    std::ranges::reverse_copy(numbers, outStream);
24 }
```

The output:

```

one; two; three; four; five; six;
six; five; four; three; two; one;
Ranges
one; two; three; four; five; six;
six; five; four; three; two; one;
```

As you can see, the ranges version is super simple to use.

See more `ranges::reverse` (<https://en.cppreference.com/w/cpp/algorithm/ranges/reverse>) and `ranges::reverse_copy` (https://en.cppreference.com/w/cpp/algorithm/ranges/reverse_copy).

55.2.8 rotate

This time let's work with words and try to rotate them around:

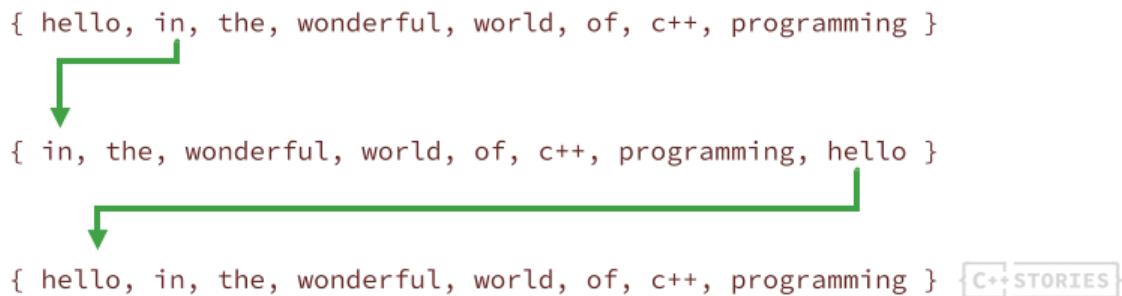
```

1 // https://godbolt.org/z/jT5dfj7bo
2 #include <algorithm>
3 #include <vector>
4 #include <ranges>
5 #include <iostream>
6 #include <iomanip>
7
8 int main() {
9     std::vector<std::string> words { "hello", "in", "the",
10         "wonderful", "world", "of", "c++", "programming",
11     };
12
13     std::ostream_iterator<std::string> out(std::cout, " ");
14
15     // standard version:
16     std::ranges::copy(words, out);
17     std::cout << '\n';
```

```

18     auto firstWord = words[0];
19     auto newPos = std::rotate(begin(words), std::next(begin(words)), 1), end(words));
20     std::ranges::copy(words, out);
21     std::cout << '\n';
22     std::cout << std::quoted(firstWord) << " is now at pos "
23                     << std::distance(begin(words), newPos) << '\n';
24
25 // ranges version:
26 auto helloPos = std::ranges::find(words, "hello");
27 if (helloPos != end(words)) {
28     auto firstWord = words[0];
29     auto ret = std::ranges::rotate(words, helloPos);
30     std::ranges::copy(words, out);
31     std::cout << '\n';
32     std::cout << std::quoted(firstWord) << " is now at pos "
33                     << std::distance(begin(words), ret.begin()) << '\n';
34 }
35 }
```

The example starts from a sentence and rotates it so that the word "the" is now the first word. Later in the ranges version, we try to find the first word of the initial sentence, and then we shift it again to get to the start.



The output:

```

hello in the wonderful world of c++ programming
in the wonderful world of c++ programming hello
"hello" is now at pos 7
hello in the wonderful world of c++ programming
"in" is now at pos 1
```

See more `ranges::rotate` (<https://en.cppreference.com/w/cpp/algorithm/ranges/rotate>).

55.2.9 shuffle

As a reminder, `std::random_shuffle` was deprecated and removed in C++17. Since C++11, it's best to use `std::shuffle` or `std::ranges::shuffle` that takes a random generator object as a parameter rather than relying on `rand()`.

Let's have a look at the basic example:

```

1 // https://godbolt.org/z/bbMGdT15r
2 #include <iostream>
3 #include <random>
4 #include <iterator>
5 #include <algorithm>
6 #include <ranges>
7
8 int main() {
9     std::vector<std::string> words {
10         "box", "tv", "car", "bricks", "game", "ball"
11     };
12
13     std::mt19937 rng{std::random_device{}()};
14
15     auto print = [] (std::string_view str, const auto& cont) {
16         std::cout << str << ":" ;
17         for (const auto &w : cont)
18             std::cout << w << ", ";
19         std::cout << '\n';
20     };
21
22     print("before", words);
23
24     // the standard version:
25     std::shuffle(begin(words), end(words), rng);
26     print("after ", words);
27
28     // the ranges version:
29     // the standard version:
30     std::ranges::shuffle(words, rng);
31     print("after ", words);
32 }
```

See more `ranges::shuffle` (<https://en.cppreference.com/w/cpp/algorithm/ranges/shuffle>).

55.2.10 sample

`std::sample` is a relatively new algorithm available since C++17. It allows you to select `n` items at

random (uniform probability) from a sequence. It's not `constexpr`. Let's see an example:

```

1 // https://godbolt.org/z/Wjnx9jjd
2 #include <iostream>
3 #include <random>
4 #include <iterator>
5 #include <algorithm>
6 #include <ranges>
7
8 struct Product {
9     std::string name_;
10    double value_ { 0.0 };
11};
12
13 int main() {
14     const std::vector<Product> prods {
15         { "box", 10.0 }, {"tv", 100.0}, {"ball", 30.0},
16         { "car", 1000.0 }, {"toy", 40.0}, {"cake", 15.0},
17         { "book", 45.0}, {"PC game", 35.0}, {"wine", 25}
18     };
19
20     std::mt19937 rng{std::random_device{}()};
21     const size_t firstRoundCount = 4;
22     const size_t secondRoundCount = 2;
23
24     // the standard version:
25     std::vector<Product> selected;
26     std::sample(begin(prods), end(prods),
27                 std::back_inserter(selected),
28                 firstRoundCount, rng);
29
30     std::cout << firstRoundCount << " selected products: \n";
31     for (const auto &elem : selected)
32         std::cout << elem.name_ << '\n';
33
34     // the ranges version:
35     std::vector<Product> onlyTwo;
36     std::ranges::sample(selected,
37                         std::back_inserter(onlyTwo),
38                         secondRoundCount, rng);
39
40     std::cout << secondRoundCount << " winners: \n";
41     for (const auto &elem : onlyTwo)
```

```

42     std::cout << elem.name_ << '\n';
43 }
```

See more `ranges::sample` (<https://en.cppreference.com/w/cpp/algorithm/ranges/sample>).

55.2.11 unique

The `unique()` algorithm allows you to clean up a consecutive group of equivalent elements. For example, from `{1, 1, 5, 5, 2, 2, 3, 3, 4, 4, 5, 5}` you may want to remove all duplicates and get `{1, 5, 2, 3, 4, 5}`. Please note that not all `5`'s were removed, only those in the same “group”.

Let's have a look at the following sample where I want to remove such duplicates:

```

1 #include <iostream>
2 #include <random>
3 #include <algorithm>
4 #include <ranges>
5
6 struct Product {
7     std::string name_;
8     double value_ { 0.0 };
9 };
10
11 int main() {
12     std::vector<Product> prods {
13         { "box", 20.0}, {"box", 10.0}, {"toy", 35.0},
14         { "box", 10.0}, {"tv", 100.0}, {"tv", 30.0},
15         { "car", 1000.0}, {"box", 0.0}, {"toy", 40.0}, {"cake", 15.0},
16     };
17
18     auto print = [] (std::string_view str, const std::vector<Product>& cont) {
19         std::cout << str << ":" ;
20         for (const auto &p : cont)
21             std::cout << p.name_ << ", ";
22         std::cout << '\n';
23     };
24
25     print("before:      ", prods);
26     auto ret = std::ranges::unique(prods, {}, &Product::name_);
27     prods.erase(ret.begin(), ret.end());
28     print("after unique: ", prods);
29     std::ranges::sort(prods, {}, &Product::name_);
30     print("after sort:   ", prods);
31     ret = std::ranges::unique(prods, {}, &Product::name_);
32     prods.erase(ret.begin(), ret.end());
```

```

33     print("another unique:", prods);
34 }
```

The output:

```

before:      : box, box, toy, box, tv, tv, car, box, toy, cake,
after unique: : box, toy, box, tv, car, box, toy, cake,
after sort:   : box, box, box, cake, car, toy, toy, tv,
another unique:: box, cake, car, toy, tv,
```

As you can see, this example didn't cover the standard version and only focused on the `ranges::unique`.

After the first run to `unique()`, the `prods` vector is modified so that elements to be removed are passed to the end of the container. What's more, they are of unspecified value. That's why I used `erase` to remove those elements from the container. The `ret` object contains a sub-range pointing to the first "removed" element and the end of the input range.

After the first "iteration," there are still some duplicated elements, but they don't share the same "group". To fix this, we can sort elements (I'm using a projection to look only at the `name_data` member). After all, elements are sorted, we can clean up the rest of duplicates. Of course, you can do the sorting before the whole cleanup.

See more `ranges::unique` (<https://en.cppreference.com/w/cpp/algorithm/ranges/unique>).

55.2.12 Summary

Wow, we covered a lot of excellent algorithms!

As you can see, with the ranges versions, you can simplify code and pass the whole sequence, the entire container. In many cases, this results in a much easier-to-read code.

Stay tuned for the next section, where I'll cover sorting algorithms, binary search, and others... and we'll have a glace of what's coming in C++23 regarding new algorithms.

55.3 sorting, sets, other and C++23 updates

This section is the third and the last one in the mini-series about ranges algorithms. We'll look at some sorting, searching, and remaining algorithms. We'll also have a glimpse of cool C++23 improvements in this area.

Let's go.

55.3.1 Partitioning & Sorting

55.3.1.1 `sort` and `is_sorted`

The `Sort`ing algorithm often comes as an advertisement for ranges. If you have a container, then thanks to ranges, you can write:

```
std::ranges::sort(myContainer);
```

See the example for a better overview:

```
1 // https://godbolt.org/z/zrnee1PMe
2 #include <iostream>
3 #include <algorithm>
4 #include <ranges>
5 #include <vector>
6
7 struct Product {
8     std::string name;
9     double value { 0.0 };
10 };
11
12 void print(std::string_view intro, const std::vector<Product>& container) {
13     std::cout << intro << '\n';
14     for (const auto &elem : container)
15         std::cout << elem.name << ", " << elem.value << '\n';
16 }
17
18 int main() {
19     const std::vector<Product> prods {
20         { "box", 10.0 }, {"tv", 100.0}, {"ball", 30.0},
21         { "car", 1000.0 }, {"toy", 40.0}, {"cake", 15.0},
22         { "book", 45.0}, {"pc game", 35.0}, {"wine", 25}
23     };
24
25     print("input", prods);
26
27     // the standard version:
28     std::vector<Product> copy = prods;
29     std::sort(begin(copy), end(copy), [](const Product& a, const Product& b)
30             { return a.name < b.name; })
31     ;
32
33     print("after sorting by name", copy);
34
35     // the ranges version:
36     copy = prods;
37     std::ranges::sort(copy, {}, &Product::name);
38     print("after sorting by name", copy);
39     std::ranges::sort(copy, {}, &Product::value);
40     print("after sorting by value", copy);
41     auto sorted = std::ranges::is_sorted(copy, {}, &Product::value);
42     std::cout << "is sorted by value: " << sorted << '\n';
```

```
43 }
```

In many implementations, the Introsort (see <https://en.wikipedia.org/wiki/Introsort>) is used. It's a hybrid solution with usually a quick sort/heap sort and then insertion sort for small (sub)ranges.

Other versions of sorting algorithms:

- `partial_sort` - sorts the first N elements of a range.
- `stable_sort` - the order of equivalent elements is stable, i.e. guaranteed to be preserved.

As you can see, with the ranges version, it's straightforward to pass a projection and sort by a given sub-part of the element. In the regular version, you need a separate lambda...

Read more at `ranges::sort` (<https://en.cppreference.com/w/cpp/algorithm/ranges/sort>).

55.3.1.2 partition

Partitioning is an essential part of quick sort. For a given predicate, the operation moves elements matching the predicate to the first part of the container and non-matching to the second part. Sometimes, you might partition a container rather than perform the full sorting operation. Have a look at the following example:

```
1 // https://godbolt.org/z/MGMrn6Kon
2 #include <iostream>
3 #include <algorithm>
4 #include <ranges>
5 #include <vector>
6
7 void print(std::string_view intro, const std::vector<auto>& container) {
8     std::cout << intro << '\n';
9     for (const auto &elem : container)
10         std::cout << elem << ", ";
11     std::cout << '\n';
12 }
13
14 int main() {
15     const std::vector vec { 11, 2, 3, 9, 5, 4, 3, 8, 4, 1, 11, 12, 10, 4 };
16
17     print("input", vec);
18
19     // the standard version:
20     auto copy = vec;
21     auto it = std::partition(begin(copy), end(copy), [](int a)
22     { return a < 7; })
23     ;
24
25     print("partition till 7", copy);
```

```

26     std::cout << "pivot at " << std::distance(begin(copy), it) << '\n';
27
28     // ranges version:
29     copy = vec;
30     auto sub = std::ranges::partition(copy, [](int a)
31         { return a < 7; })
32     );
33
34     print("partition till 7", copy);
35     std::cout << "pivot at " << std::distance(begin(copy), sub.begin()) << '\n';
36 }
```

The output:

```

input
11, 2, 3, 9, 5, 4, 3, 8, 4, 1, 11, 12, 10, 4,
partition till 7
4, 2, 3, 1, 5, 4, 3, 4, 8, 9, 11, 12, 10, 11,
pivot at 8
partition till 7
4, 2, 3, 1, 5, 4, 3, 4, 8, 9, 11, 12, 10, 11,
pivot at 8
```

As you can see, we could easily divide the container into two groups: the first part contains elements smaller than 7, and the second part with elements ≥ 7 . The relative order between elements might be altered (you need `stable_partition` to keep that order).

The interface for `partition` is relatively simple. The ranges version additionally takes a projection, but the example didn't use it. One difference is that `ranges::partition` returns a subrange rather than an iterator (as with the `std::` version).

See more about the algorithms at `ranges::is_partitioned` (https://en.cppreference.com/w/cpp/algorithm/ranges/is_partitioned) and `ranges::partition` (<https://en.cppreference.com/w/cpp/algorithm/ranges/partition>).

55.3.2 Binary Search operations

If your container is already sorted, then you can perform logarithmic binary search operations.

55.3.2.1 `binary_search`

```

1 // https://godbolt.org/z/EPv7hr3nq
2 #include <iostream>
3 #include <algorithm>
4 #include <ranges>
5 #include <vector>
6 #include <numeric>
7
8
```

```

9 void print(std::string_view intro, const auto& container) {
10    std::cout << intro << '\n';
11    for (const auto &elem : container)
12        std::cout << elem << ", ";
13    std::cout << '\n';
14 }
15
16 int main() {
17    std::vector<int> vec(100, 0);
18    std::iota(begin(vec), end(vec), 0);
19
20    print("first ten elements of input", vec | std::views::take(10));
21
22    // the standard version:
23    auto copy = vec;
24    auto found = std::binary_search(begin(copy), end(copy), 13);
25    std::cout << "found 13: " << found << '\n';
26
27    // ranges version:
28    copy = vec;
29    found = std::ranges::binary_search(copy, 13);
30    std::cout << "found 13: " << found << '\n';
31 }
```

See more at `ranges::binary_search` (https://en.cppreference.com/w/cpp/algorithm/ranges/binary_search). Additionally you can use related algorithms:

- `std::ranges::lower_bound` (https://en.cppreference.com/w/cpp/algorithm/ranges/lower_bound) - returns an iterator to the first element not less than the given value
- `std::ranges::upper_bound` (https://en.cppreference.com/w/cpp/algorithm/ranges/upper_bound) - returns an iterator to the first element greater than a certain value

55.3.3 Set operations

There are many set-related functions in the library some of them:

- `ranges::merge` - merges two sorted ranges
- `ranges::inplace_merge` - merges two ordered ranges in-place
- `ranges::includes` - returns true if one sorted sequence is a subsequence of another sorted sequence
- `ranges::set_difference` - computes the difference between two sets
- `ranges::set_intersection` - computes the intersection of two sets
- `ranges::set_symmetric_difference` - computes the symmetric difference between two sets

- `ranges::set_union` - computes the union of two sets

As an example let's have a look at one case with `includes`:

55.3.3.1 includes

Returns `true` if the sorted range is a subsequence of another sorted range.

```

38     );
39     std::cout << "contains the name set: " << ret << '\n';
40
41
42     // the ranges version:
43     ret = std::ranges::includes(prods, namesToCheck, {}, &Product::name);
44     std::cout << "contains the name set: " << ret << '\n';
45 }
```

The ranges version is simpler and offers a way to check against different containers. With the `std::` approach, the iterator needs to be dereferenced and then implicitly converted to both input container element types.

See more at `std::includes` (<https://en.cppreference.com/w/cpp/algorithm/includes>).

55.3.4 Other

55.3.4.1 max_element

Searching for the max element in a container (unsorted):

```

1 // https://godbolt.org/z/6KW198x5T
2 #include <iostream>
3 #include <random>
4 #include <iterator>
5 #include <algorithm>
6 #include <ranges>
7
8 struct Product {
9     std::string name_;
10    double value_ { 0.0 };
11};
12
13 int main() {
14     const std::vector<Product> prods {
15         { "box", 10.0 }, {"tv", 100.0}, {"ball", 30.0},
16         { "car", 1000.0 }, {"toy", 40.0}, {"cake", 15.0},
17         { "book", 45.0}, {"PC game", 35.0}, {"wine", 25}
18     };
19
20     // the standard version:
21     auto res = std::max_element(begin(prods), end(prods),
22         [] (const Product& a, const Product& b) {
23             return a.value_ < b.value_;
24         });
25 }
```

```

26     if (res != end(prods)) {
27         const auto pos = std::distance(begin(prods), res);
28         std::cout << "std::max_element at pos " << pos
29                         << ", val " << res->value_ << '\n';
30     }
31
32     // the ranges version:
33     auto it = std::ranges::max_element(prods, {}, &Product::value_);
34     if (it != end(prods)) {
35         const auto pos = std::distance(begin(prods), it);
36         std::cout << "std::max_element at pos " << pos
37                         << ", val " << res->value_ << '\n';
38     }
39 }
```

55.3.4.2 equal

```

1 // https://godbolt.org/z/fb4cMsh4P
2 #include <iostream>
3 #include <random>
4 #include <iterator>
5 #include <algorithm>
6 #include <ranges>
7
8 struct Product {
9     std::string name;
10    double value { 0.0 };
11};
12
13 int main() {
14     const std::vector<Product> prods {
15         { "box", 10.0 }, {"tv", 100.0}, {"ball", 30.0},
16         { "car", 1000.0 }, {"toy", 40.0}, {"cake", 15.0},
17     };
18
19     const std::vector<Product> moreProds {
20         { "box", 11.0 }, {"tv", 120.0}, {"ball", 30.0},
21         { "car", 10.0 }, {"toy", 39.0}, {"cake", 15.0}
22     };
23
24     // the standard version:
25     auto res = std::equal(begin(prods), end(prods),
26                           begin(moreProds), end(moreProds),
27                           // comparison function
28                           // ...
29                           );
30 }
```

```

27     [](const Product& a, const Product& b) {
28         return a.name == b.name;
29     };
30
31     std::cout << "equal: " << res << '\n';
32
33 // the ranges version:
34 res = std::ranges::equal(prods, moreProds, {}, &Product::name, &Product::name);
35 std::cout << "equal: " << res << '\n';
36 }
```

See more at `ranges::equal` (<https://en.cppreference.com/w/cpp/algorithm/ranges/equal>).

55.3.4.3 Even more

My list of algorithms is not complete. Almost all standard algorithms have their `std::ranges::` alternative. Have a look at the following interesting algorithms that haven't been mentioned in the series:

Heap operations:

- `ranges::is_heap`
- `ranges::is_heap_until`
- `ranges::make_heap`
- `ranges::push_heap`
- `ranges::pop_heap`
- `ranges::sort_heap`

Permutations:

- `ranges::is_permutation`
- `ranges::next_permutation`
- `ranges::prev_permutation`

Uninitialized memory algorithms:

- `ranges::uninitialized_copy`
- `ranges::uninitialized_copy_n`
- `ranges::uninitialized_fill`
- `ranges::uninitialized_fill_n`
- `ranges::uninitialized_move`
- `ranges::uninitialized_move_n`
- `ranges::uninitialized_default_construct`

- `ranges::uninitialized_default_construct_n`
- `ranges::uninitialized_value_construct`
- `ranges::uninitialized_value_construct_n`
- `ranges::destroy`
- `ranges::destroy_n`
- `ranges::destroy_at`
- `ranges::construct_at`

55.3.5 Numeric

As of C++20, we have most of the corresponding ranges algorithms from the `<algorithm>` header, but the `<numeric>` header is missing.

55.3.6 Soon in C++23

C++23 specification is almost complete and in the feature-freeze mode. So far I'm aware of the following algorithms that we'll land in the new C++ version:

- `ranges::starts_with` and `ranges::ends_with` (as of June 2022 available in the MSVC compiler)
- `ranges::contains` (P2302)
- `ranges::shift_left` and `ranges::shift_right`,
- `ranges::iota`
- `ranges::fold` - as an alternative for `std::accumulate`

55.3.7 Summary

This article completes our journey through most C++ algorithms available in the Standard Library (except for numerics). Most of the algorithms have their `ranges::` counterparts, and in C++23, we'll have even more additions.

Chapter 56

New Fold Algorithms

• Sy Brand 📅 2023-04-13 🔍 ★★

C++20 added new versions of the standard library algorithms which take ranges as their first argument rather than iterator pairs, alongside other improvements. However, key algorithms like `std::accumulate` were not updated. This has been done in C++23, with the new `std::ranges::fold_*` family of algorithms. The standards paper for this is P2322 and was written by Barry Revzin. It been implemented in Visual Studio 2022 version 17.5. In this post I'll explain the benefits of the new “rangified” algorithms, talk you through the new C++23 additions, and explore some of the design space for fold algorithms in C++.

56.1 Background: Rangified Algorithms

C++20's algorithms make several improvements to the old iterator-based ones. The most obvious is that they now can take a range instead of requiring you to pass iterator pairs. But they also allow passing a “projection function” to be called on elements of the range before being processed, and the use of C++20 concepts for constraining their interfaces more strictly defines what valid uses of these algorithms are. These changes allow you to make refactors like:

```
1 // C++17 algorithm
2 cat find_kitten(const std::vector<cat>& cats) {
3     return *std::find_if(cats.begin(), cats.end(),
4         [](cat const& c) { return c.age == 0; });
5 }
6
7 // C++20 algorithm
8 cat find_kitten(std::span<cat> cats) {
9     return *std::ranges::find(cats, 0, &cat::age);
10 }
```

The differences here are:

1. Instead of having to pass `cats.begin()` and `cats.end()`, we just pass `cats` itself.
2. Since we are comparing a member variable of the `cat` to `0`, in C++17 we need to use `std::find_if` and pass a closure which accesses that member and does the comparison. Since the rangified

algorithms support projections, in C++20 we can use `std::ranges::find` and pass `&cat::age` as a projection, getting rid of the need for the lambda completely.

These improvements can greatly clean up code which makes heavy use of the standard library algorithms.

Unfortunately, alongside the algorithms which reside in the `<algorithm>` header, there are also several important ones in the `<numeric>` header, and these were not ratified in C++20¹. In this post we’re particularly interested in `std::accumulate` and `std::reduce`.

56.2 accumulate and reduce

`std::accumulate` and `std::reduce` are both fold² operations. They “fold” or “reduce” or “combine” multiple values into a single value. Both take two iterators, an initial value, and a binary operator (which defaults to `+`). They then run the given operator over the range of values given by the iterators, collecting a result as they go. For instance, given `std::array<int, 3> arr = {1, 2, 3}`, `std::accumulate(begin(arr), end(arr), 0, std::plus())` will run $((0 + 1) + 2) + 3$. Or `std::accumulate(begin(arr), end(arr), 0, f)` will run $f(f(f(0, 1), 2), 3)$.

These functions are both what are called *left folds* because they run from left to right. There’s also *right folds*, which as you may guess, run from right to left. For the last example a right fold would look like `f(1, f(2, f(3, 0)))`. For some operations, like `+`, these would give the same result, but for operations which are not associative³ (like `-`), it could make a difference.

So why do we have both `std::accumulate` and `std::reduce`? `std::reduce` was added in C++17 as one of the many parallel algorithms⁴ which let you take advantage of parallel execution for improved performance. The reason it has a different name than `std::accumulate` is because it has different constraints on what types and operations you can use: namely the operation used must be both associative and commutative⁵.

To understand why, consider the following code:

```

1 std::array<int, 8> a {0, 1, 2, 3, 4, 5, 6, 7};
2 auto result = std::reduce(std::execution::par, //execute in parallel
3 begin(a), end(a), 0, f);

```

If `f` is associative, then `std::reduce` could reduce the first half of `a` on one CPU core, reduce the second half of `a` on another core, then call `f` on the result. However, if `f` is not associative, this could result in a different answer than carrying out a pure left fold. As such, requiring associativity lets `std::reduce` distribute the reduction across multiple units of execution.

Requiring commutativity likewise enables some potential optimizations. One is that the implementation is free to reorder operations as it likes. If, for example, some calls to `f` take longer than others, `reduce` could use whichever intermediate results are available without having to wait around for the “right” result. Commutativity also enables vectorization when using the `std::execution::par_unseq` policy.

¹ Why? Quoting Casey Carter: “Every time someone asks why we didn’t cover `<numeric>` and `<memory>` algorithms: We thought 187 pages of Technical Specification was enough.”

²[https://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))

³https://en.wikipedia.org/wiki/Associative_property

⁴<https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/>

⁵https://en.wikipedia.org/wiki/Commutative_property

For example, if `f` is addition, the first half of `a` could be loaded into one vector register, the second half loaded into another, and a vector addition executed on them. This would result in $(0 + 4) + (1 + 5) + (2 + 6) + (3 + 7)$. Notice that the operands have been interleaved: this requires commutativity.

56.3 std::ranges::fold_*

C++23 comes with six fold functions which fulfil different important use cases. The one you'll reach for most is `std::ranges::fold_left`.

56.3.1 fold_left

You can use `fold_left` in place of calls to `std::accumulate`. For instance, I have three cats, and when I brush them, I collect all the loose fur as I go so I can throw it away:

```

1 fur brush(fur existing_fur, cat& c);
2 std::vector<cat> cats = get_cats();
3
4 // C++20
5 auto loose_fur = std::accumulate(begin(cats), end(cats), fur{}, brush);
6 // C++23
7 auto loose_fur = std::ranges::fold_left(cats, fur{}, brush);
8
9 throw_away(loose_fur);

```

`std::accumulate` is really a generic left fold, but its name suggests summation, and the defaulting of the binary operator to addition further contributes to this. This makes uses of `accumulate` for non-summation purposes look a little clunky. This is why the new version is instead called `fold_left`, and does not have a default operator.

56.3.2 fold_right

As you can probably guess, since there's a `fold_left` function, there's also a `fold_right` function. For associative operations like `brush`, there's no real difference in behaviour. But say we have a function which takes some amount of food and feeds half of it to a cat, returning the leftovers:

```

1 food feed_half(cat& c, food f) {
2     auto to_feed = f / 2;
3     c.eaten += to_feed;
4     return f - to_feed;
5 }

```

This operation is not associative, therefore using `fold_left` or `fold_right` would result in feeding different cats different amounts of food. We could call `fold_right` like this:

```

1 std::vector<cat> cats = get_cats();
2 //feed cats from right to left, starting with 100 food
3 auto leftovers = std::ranges::fold_right(cats, 100, feed_half);

```

Note that for `fold_right`, the order of arguments to the operator are flipped from `fold_left`: the accumulator is on the right rather than the left.

In these examples we've been able to pick a reasonable initial value for our folds (`fur{}` in the first, `100` in the second). But how do we pick our initial element? What if there is no good initial value to pick?

56.3.3 Aside on initial element

`std::reduce` allows omitting the initial element for the fold, in which case it uses a value-initialized object of the given iterator's value type (e.g. `0` for `int`). This can be handy in some cases, such as summing integers, but doesn't work so well in others, such as taking the product of integers. Usually what we want for the initial element is some identity element for the value type of the range with respect to the given binary operator. Given any object `x` of type `T` and operation `f`, the identity element `id` is one for which `f(id, x) == x`. For example, the identity element for the pair `int`, `operator+` is `0`. For `int`, `operator*` it's `1`. For `std::string`, `operator+` it's `" "`.

These pairs of types and associative binary operators which have an identity element turn out to be surprisingly common in programming, they're called monoids⁶. Ben Deane has several great talks on monoids in C++, I'd highly recommend watching this one⁷.

We don't have a way to easily get at the identity element of a given monoid in C++. You could imagine implementing some `monoid_traits` class template which could be specialised by users to support custom types. This could let `fold_left` be implemented something like (omitting constraints for brevity):

```
1 template <class Rng, class F, class T = std::ranges::range_value_t<Rng>>
2 constexpr T fold_left (Rng&& rng, F&& op, T init =
3     monoid_traits<std::ranges::range_value_t<Rng>, F>::identity_element());
```

Maybe you think that's horrifying and too much work. I might agree for many cases. I do think it's an interesting design area though, and I know GraphBLAS⁸ does something like this; there's a CppCon⁹ talk which shows the kinds of things they do with monoid traits. P1813 explored a similar idea for a concepts design for numeric algorithms.

I think the C++23 design makes the right decision in simply not defaulting the initial element at all. This forces you to think about it and not accidentally supply a value-initialized value in cases where it's not correct.

56.3.4 `fold_left_first` and `fold_right_last`

But what if you don't have an identity element for your type? Then you need to pull out the first element by-hand, which is quite unwieldy and a bit tricky to get right for input iterators¹⁰:

```
1 auto b = std::ranges::begin(rng);
2 auto e = std::ranges::end(rng);
3 auto init = *b;
4 fold_left(std::ranges::next(b), e, std::move(init), f);
```

⁶<https://en.wikipedia.org/wiki/Monoid>

⁷<https://www.youtube.com/watch?v=INnattuluiM>

⁸<https://graphblas.org/>

⁹<https://www.youtube.com/watch?v=xMBNCtFV8sI>

¹⁰https://en.cppreference.com/w/cpp/iterator/input_iterator

As such, many languages provide an alternative `fold` function which uses the first element of the range as the initial element (thus additionally requiring the range to be non-empty).

The version we have in C++23 has this too, it calls them `fold_left_first` and `fold_right_last`. This lets you simply write:

```
std::ranges::fold_left_first(rng, f);
```

Much better.

56.3.5 `fold_left_with_iter` and `fold_left_first_with_iter`

The final two versions of `fold` which are in C++23 are ones which expose an additional result computed by the fold: the end iterator. Recall that for some ranges, the type returned by `std::ranges::end` is not an iterator, but a sentinel: some rule for when to finish iteration. For some ranges, computing the iterator for the range which is equal to what `std::ranges::end` returns may actually be quite expensive. Since carrying out the fold necessarily requires computing this iterator, C++23 provides functions which return this iterator alongside the value computed.

For example, say we have a collection of cats sorted by age, and we have some food which is specially formulated for younger cats. We could split the food between the younger cats like so:

```
1 std::vector<cat> cats = get_sorted_cats();
2 auto young_cats = cats | std::views::take_while([](auto c) { return c.age < 7; });
3 auto leftover_food = std::ranges::fold_left(young_cats, food_for_young_cats, feed_half);
```

However, now if we want to give some other food to the older cats, we need to recompute the point in cats where the young cats stop and the older cats begin:

```
1 auto first_old_cat = std::ranges::find_if(cats, [](auto c) { return c.age >= 7; });
2 give_some_other_food(first_old_cat, std::ranges::end(cats));
```

`fold_left_with_iter` lets you avoid the recomputation:

```
1 std::vector<cat> cats = get_sorted_cats();
2 auto young_cats = cats | std::views::take_while([](auto c) { return c.age < 7; });
3 auto [first_old_cat, leftover_food]
4     = std::ranges::fold_left_with_iter(young_cats, food_for_young_cats, feed_half);
5
6 give_some_other_food(first_old_cat, std::ranges::end(cats));
```

56.4 What about reduce?

The `fold` family of functions extend and replace `std::accumulate`. But what about `std::reduce`? `std::ranges::reduce` is planned, but no one has written the necessary standards proposal for it yet. It wouldn't be that different from `std::ranges::fold_*`, but there's some subtle design points, like different constraints on the binary operator to allow `op(*it, acc)`, `op(acc, *it)`, and `op(*it, *it)`.

56.5 What about projections?

I said at the start that the ranged algorithms have 3 main benefits:

1. Can pass a range rather than iterator pair
2. Are constrained by concepts
3. Support projection functions

Only the first two apply for `fold_*`, however: projection functions aren't supported for a rather subtle reason. You can see P2322r6 for all the details, but essentially, for the `fold_left_first*` and `fold_right_last*` overloads, allowing projections would incur an extra copy even in cases where it shouldn't be required. As such, projections were removed from those overloads, and then from the remaining three for consistency.

If you want to use a projection function with `fold_left`, you could do something like:

```
auto res = fold_left(rng | std::views::transform(projection), init, f);
```

56.6 Feedback

You can try out the feature in Visual Studio 2022 version 17.5. If you have any questions, comments, or issues with the feature, you can reach us via email at visualcpp@microsoft.com or via Twitter at @VisualC.

56.7 Acknowledgements

Thanks to Christopher Di Bella, Barry Revzin, and Stephan T. Lavavej for feedback and information.
Thanks to Jakub Mazurkiewicz for implementing this feature in our standard library.

Chapter 57

Faster integer formatting - James Anhalt (jeaiiii)'s algorithm

👤 Junekey Jeon 📅 2022-02-16 🏷 ★★★★☆

This post is about an ingenious algorithm for printing integers into decimal strings. It sounds like an extremely simple problem, but it is in fact quite more complicated than one might imagine. Let us more precisely define what we want to do: we take an integer of specific bit-width and a byte buffer, and convert the input integer into a string consisting of its decimal digits, and then write it into the given buffer. For simplicity, we will assume that the integer is unsigned and is of 32-bits. So, we want to implement the following function written in C++:

```
1 char* itoa(std::uint32_t n, char* buffer) {
2     // Convert n into decimal digit string and write it into buffer.
3     // Returns the position right next to the last character written.
4 }
```

There are numerous algorithms for doing this, and I will dig into a clever algorithm invented by James Anhalt (jeaiiii)¹, which seems to be the fastest known algorithm at the point of writing this post.

57.1 Disclaimer

I actually have not looked (and will not look) carefully at his code (MACROS², oh my god) and have no idea what precisely was the method of analysis he had in mind. All I write here is purely my own analysis inspired by reading these lines of comment he wrote:

```
1 // 1. form a 7.32 bit fixed point number: t = u * 2^32 / 10^log10(u)
2 // 2. convert 2 digits at a time [00, 99] by lookup from the integer portion of
3 //     the fixed point number (the upper 32 bits)
4 // 3. multiply the fractional part (the lower 32 bits) by 100 and repeat until
5 //     1 or 0 digits left
```

¹<https://github.com/jeaiiii/itoa>

²https://github.com/jeaiiii/itoa/blob/main/itoa/itoa_jeaiiii.cpp

```

6 // 4. if 1 digit left multipy by 10 and convert it (add '0')
7 //
8 // N == log10(u)
9 // finding N by binary search for a 32bit number N = [0, 9]
10 // this is fast and selected such 1 & 2 digit numbers are faster (2 branches) than
11 // long numbers (4 branches) for the 10 cases
12 //
13 //      \-----_
14 //      / \----- \-----
15 //      \ \   \ \   \ \   \
16 // 0 1 / \ / \ / \ / \
17 // 2 3 4 5 6 7 8 9

```

So it is totally possible that in fact what's written in this post has nothing to do with what he actually did; however, I strongly believe that what I ended up with is more or less equivalent to what his code is doing, modulo some small minor differences.

57.2 Naïve implementations

The very first problem that anyone who tries to implement such a function will face is that we want to write digits from left to right, but naturally we compute the digits from right to left. Hence, unless we know the number of decimal digits in the input upfront, we do not know the exact position in the buffer that we can write our digits into. There are several different strategies to cope with this issue. Probably the simplest (and quite effective) one is to just print the digit from right to left but into a temporary buffer, and after we get all digits of [Math Processing Error] we copy the temporary buffer back to the destination buffer. At the point of obtaining the left-most decimal digit of [Math Processing Error], we also get the length of the string, so we know what exact bytes to copy.

With this strategy, we can think of the following implementation:

```

1 // https://godbolt.org/z/7G7ecs7r4
2 char* itoa_naive(std::uint32_t n, char* buffer) {
3     char temp[10];
4     char* ptr = temp + sizeof(temp) - 1;
5     while (n >= 10) {
6         *ptr = char('0' + (n % 10));
7         n /= 10;
8         --ptr;
9     }
10    *ptr = char('0' + n);
11    auto length = temp + sizeof(temp) - ptr;
12    std::memcpy(buffer, ptr, length);
13    return buffer + length;
14 }

```

The size of the temporary buffer is set to [Math Processing Error], because that's the maximum possible decimal length for `std::uint32_t`.

The mismatch between the order of computation and the order of desired output is indeed a quite nasty problem, but let us forget about this issue for a while because there is something more interesting to say here.

There are several performance issues in this code, and one of them is the division by [Math Processing Error]. Of course, since the divisor is a known constant, our lovely compiler will automatically convert the division into multiply-and-shift (see this classic paper³ for example), so we do not need to worry about the dreaded `idiv` instruction which is extremely infamous of its performance. However, for simple enough algorithms like this, multiplication can still be a performance killer, so it is reasonable to expect that we will get a better performance by reducing the number of multiplications.

Regarding this, Andrei Alexandrescu popularized the idea of generating two digits per a division, so halving the required number of multiplications. That is, we first prepare a lookup table for converting [Math Processing Error]-digit integers into strings. Then in the loop we perform divisions by [Math Processing Error], rather than [Math Processing Error], to get [Math Processing Error] digits per each division. The following code illustrates this:

```

1 // https://godbolt.org/z/vnMTf7s9r
2 static constexpr char radix_100_table[] = {
3     '0', '0', '0', '1', '0', '2', '0', '3', '0', '4',
4     '0', '5', '0', '6', '0', '7', '0', '8', '0', '9',
5     '1', '0', '1', '1', '1', '2', '1', '3', '1', '4',
6     '1', '5', '1', '6', '1', '7', '1', '8', '1', '9',
7     '2', '0', '2', '1', '2', '2', '2', '3', '2', '4',
8     '2', '5', '2', '6', '2', '7', '2', '8', '2', '9',
9     '3', '0', '3', '1', '3', '2', '3', '3', '3', '4',
10    '3', '5', '3', '6', '3', '7', '3', '8', '3', '9',
11    '4', '0', '4', '1', '4', '2', '4', '3', '4', '4',
12    '4', '5', '4', '6', '4', '7', '4', '8', '4', '9',
13    '5', '0', '5', '1', '5', '2', '5', '3', '5', '4',
14    '5', '5', '5', '6', '5', '7', '5', '8', '5', '9',
15    '6', '0', '6', '1', '6', '2', '6', '3', '6', '4',
16    '6', '5', '6', '6', '6', '7', '6', '8', '6', '9',
17    '7', '0', '7', '1', '7', '2', '7', '3', '7', '4',
18    '7', '5', '7', '6', '7', '7', '7', '8', '7', '9',
19    '8', '0', '8', '1', '8', '2', '8', '3', '8', '4',
20    '8', '5', '8', '6', '8', '7', '8', '8', '8', '9',
21    '9', '0', '9', '1', '9', '2', '9', '3', '9', '4',
22    '9', '5', '9', '6', '9', '7', '9', '8', '9', '9'
23 };
24
25 char* itoa_two_digits_per_div(std::uint32_t n, char* buffer) {
26     char temp[8];

```

³<https://gmplib.org/tege/divcnst-pldi94.pdf>

```

27     char* ptr = temp + sizeof(temp);
28     while (n >= 100) {
29         ptr -= 2;
30         std::memcpy(ptr, radix_100_table + (n % 100) * 2, 2);
31         n /= 100;
32     }
33     if (n >= 10) {
34         std::memcpy(buffer, radix_100_table + n * 2, 2);
35         buffer += 2;
36     }
37     else {
38         buffer[0] = char('0' + n);
39         buffer += 1;
40     }
41     auto remaining_length = temp + sizeof(temp) - ptr;
42     std::memcpy(buffer, ptr, remaining_length);
43     return buffer + remaining_length;
44 }
```

57.3 The core idea of James Anhalt' s algorithm

So, with the above trick of grouping 2 digits, how many multiplications do we need for integers of, say, 10 decimal digits? Note that we need to compute both the quotient and the remainder, and as far as I know at least 2 multiplications should be performed to get both of them and there is no way to do it with just one multiplication. Hence, for each pair of 2 digits, we need to perform 2 multiplications, thus for integers with 10 digits we need 8 multiplications, since we need 4 divisions to separate 5 pairs of 2 digits.

Quite surprisingly, in fact we can almost halve that number again into 5, which (I believe) is the core benefit of James Anhalt' s algorithm. The crux of the idea can be summarized as follows: given n , we find an integer y satisfying

$$n = \left\lfloor \frac{10^k y}{2^D} \right\rfloor \quad (57.1)$$

for some nonnegative integer constants k and D .

This transformation is a real deal, because after we get such y , we can extract 2 digits of n per a multiplication. To see how, recall that in general

$$\left\lfloor \frac{a}{bc} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{b} \right\rfloor}{c} \right\rfloor \quad (57.2)$$

holds for any positive integers a,b,c ; that is, dividing a by bc is equivalent to dividing a by b first and then by c . Therefore, for any $l \leq k$, we have

$$\left\lfloor \frac{10^{k-l} y}{2^D} \right\rfloor = \left\lfloor \frac{n}{10^l} \right\rfloor \quad (57.3)$$

So, for example, let $k = l = 8$, then

$$\left\lfloor \frac{n}{10^8} \right\rfloor = \left\lfloor \frac{y}{2^D} \right\rfloor \quad (57.4)$$

Assuming that n is of 10 digits, the left-hand side is precisely the first 2 digits of n , while the right-hand side is just the right-shift of y by D -bits.

On the other hand, the next 2 digits of n can be computed as

$$\left(\left\lfloor \frac{n}{10^6} \right\rfloor \mod 10^2 \right) = \left(\left\lfloor \frac{10^2 y}{2^D} \right\rfloor \mod 10^2 \right). \quad (57.5)$$

Note that, if we write y as $y = 2^D q + r$ where q is the quotient and r is the remainder, then

$$10^2 y = 2^D (10^2 q) + 10^2 r, \quad (57.6)$$

so

$$\left(\left\lfloor \frac{10^2 y}{2^D} \right\rfloor \mod 10^2 \right) = \left(\left(10^2 q + \left\lfloor \frac{10^2 r}{2^D} \right\rfloor \right) \mod 10^2 \right) = \left(\left\lfloor \frac{10^2 r}{2^D} \right\rfloor \mod 10^2 \right) \quad (57.7)$$

Also, since $r < 2^D$, $\left\lfloor \frac{10^2 r}{2^D} \right\rfloor$ is strictly less than 10^2 , so we get

$$\left(\left\lfloor \frac{n}{10^6} \right\rfloor \mod 10^2 \right) = \left\lfloor \frac{10^2 r}{2^D} \right\rfloor. \quad (57.8)$$

This means that, in order to compute the next 2 digits of n , we first obtain the remainder of y divided by 2^D , and then multiply 10^2 to it, and then obtain the quotient of it divided by 2^D . In other words, we just need to first obtain the lowest D -bits, multiply 10^2 to it, and then right-shift the result by D -bits. As you can see, we only need 1 multiplication here.

This trend continues: to compute the next 2 digits of n , we only need to obtain the lowest D -bits, multiply 10^2 , and then right-shift the result by D -bits, so in particular we only need 1 multiplication for generating each pair of 2 digits. Indeed, it can be inductively shown that if we write $y_0 = y$ and $y_{i+1} = 10^2 (y_i \mod 2^D)$, then

$$\left(\left\lfloor \frac{n}{10^{k-2i}} \right\rfloor \mod 10^2 \right) = \left\lfloor \frac{y_i}{2^D} \right\rfloor \quad (57.9)$$

holds for each $i = 0, 1, 2, 3, 4$.

57.4 How to compute y ?

Alright, so we get that having y satisfying

$$n = \left\lfloor \frac{10^k y}{2^D} \right\rfloor \quad (57.10)$$

is pretty useful for our purpose. The next question is how to find such y . Note that the above equality is equivalent to the inequality

$$n \leq \frac{10^k y}{2^D} < n + 1, \quad (57.11)$$

or

$$\frac{2^D n}{10^k} \leq y < \frac{2^D (n + 1)}{10^k}. \quad (*)$$

Assuming $2^D \geq 10^k$, $y = \left\lceil \frac{2^D n}{10^k} \right\rceil$ will obviously do the job, but computing $\left\lceil \frac{2^D n}{10^k} \right\rceil$ might be nontrivial. Hence, let us first try something easier:

$$y = n \left\lceil \frac{2^D}{10^k} \right\rceil. \quad (57.12)$$

Here, $\left\lceil \frac{2^D}{10^k} \right\rceil$ is just a constant and is not dependent on n , so computing y is just a single integer multiplication.

With $k = 8$ and $n < 2^{32}$, we can show that this y always satisfies $(*)$ if we take $D \geq 57$; we just need to find the smallest D satisfying $(2^{32} - 1) \left(\left\lceil \frac{2^D}{10^k} \right\rceil - \frac{2^D}{10^k} \right) < \frac{2^D}{10^k}$. Hence, choosing $D = 57$, we get the magic number

$$\left\lceil \frac{2^D}{10^k} \right\rceil = 1441151881. \quad (57.13)$$

This leads us to the following code for always printing 10 digits of given n with possible leading zeros:

```

1 // https://godbolt.org/z/9c4Mb76hc
2 char* itoa_always_10_digits(std::uint32_t n, char* buffer) {
3     constexpr auto mask = (std::uint64_t(1) << 57) - 1;
4     auto y = n * std::uint64_t(1441151881);
5     std::memcpy(buffer + 0, radix_100_table + int(y >> 57) * 2, 2);
6     y &= mask;
7     y *= 100;
8     std::memcpy(buffer + 2, radix_100_table + int(y >> 57) * 2, 2);
9     y &= mask;
10    y *= 100;
11    std::memcpy(buffer + 4, radix_100_table + int(y >> 57) * 2, 2);
12    y &= mask;
13    y *= 100;
14    std::memcpy(buffer + 6, radix_100_table + int(y >> 57) * 2, 2);
15    y &= mask;
16    y *= 100;
17    std::memcpy(buffer + 8, radix_100_table + int(y >> 57) * 2, 2);
18

```

```

19     return buffer + 10;
20 }

```

The constant 1441151881 is only of 31-bits and multiplications are performed in 64-bits so there is no overflow.

57.5 Consideration of variable length

One can easily modify the above algorithm to omit printing leading decimal zeros and align the output to the left-most position of the buffer. However, the resulting algorithm is not very nice; although it only performs no more than 5 multiplications, it always performs 5 multiplications, even for short numbers like $n = 15$.

What James Anhalt did with this is complete separation of the code paths for all possible lengths of n . I mean, something like this:

```

1 //      \-----  

2 //      / \----- \-----  

3 //      \   \   \   \   \  

4 //  0  1  \  2  \  3  \  4  \  5  \  6  \  7  \  8  \  9  

5 //          2  3  4  5  6  7  8  9  

6 char* itoa_var_length(std::uint32_t n, char* buffer) {  

7     if (n < 100) {  

8         if (n < 10) {  

9             // 1 digit.  

10    }  

11    else {  

12        // 2 digits.  

13    }  

14 }  

15 else if (n < 100'0000) {  

16     if (n < 1'0000) {  

17         if (n < 1'000) {  

18             // 3 digits.  

19         }  

20         else {  

21             // 4 digits.  

22         }  

23     }  

24     else {  

25         if (n < 10'0000) {  

26             // 5 digits.  

27         }  

28         else {  

29             // 6 digits.

```

```

30     }
31   }
32 }
33 else if (n < 1'0000'0000) {
34   if (n < 1000'0000) {
35     // 7 digits.
36   }
37 else {
38   // 8 digits.
39 }
40 }
41 else if (n < 10'0000'0000) {
42   // 9 digits.
43 }
44 else {
45   // 10 digits.
46 }
47 }

```

It sounds pretty crazy, but it does the job quite well.

Now, recall that our main job is to find y satisfying

$$n = \left\lfloor \frac{10^k y}{2^D} \right\rfloor \quad (57.14)$$

Note that the choice $k = 8$ was to make sure that $\lfloor \frac{y}{2^D} \rfloor$ is the first 2 digits, given that n is of 10 digits. Since n is not always of 10 digits, we may take different k 's for each branch. For example, when n is of 3 digits, we may want to choose $k = 2$. Then since $n \leq 999$ in this case, the choice

$$y = n \left\lceil \frac{2^D}{10^k} \right\rceil \quad (57.15)$$

is valid for any $D \geq 12$, as $999 \cdot \left(\left\lceil \frac{2^{12}}{10^2} \right\rceil - \frac{2^{12}}{10^2} \right) < \frac{2^{12}}{10^2}$ holds. In this case, it is in fact better to choose $D = 32$ rather than $D = 12$, because in platforms such as x86 obtaining the lower half of a 64-bit integer is basically no-op.

As a result, the first digit of n can be computed as

$$\left\lfloor \frac{n}{10^2} \right\rfloor = \left\lfloor \frac{y}{2^{32}} \right\rfloor = \left\lfloor \frac{\lfloor 2^{32}/10^2 \rfloor n}{2^{32}} \right\rfloor = \left\lfloor \frac{42949673 \cdot n}{2^{32}} \right\rfloor, \quad (57.16)$$

and the remaining 2 digits can be computed as

$$\left(\left\lfloor \frac{n}{10^0} \right\rfloor \mod 10^2 \right) = \left\lfloor \frac{10^2 (y \mod 2^{32})}{2^{32}} \right\rfloor. \quad (57.17)$$

Similarly, we can choose $D = 32$ (with the above y with different k' s) for n' 's up to 6 digits (so $k = 0, 2, 4$), but for larger n our simplistic analysis does not allow us to do so. For n' 's with 7 or 8 digits, we set $k = 6$, and it can be shown that $D = 47$ does the job. For n' 's with 9 or 10 digits, we set $k = 8$, and as we have already seen $D = 57$ does the job. With these choices of parameters, we get the following code:

```

1 // https://godbolt.org/z/froGhEn3s
2
3 //      \-----
4 //      / \----- \-----
5 //      \ \   \ \   \ \   \
6 //  0  1  / \   / \   / \   /
7 //      2  3  4  5  6  7  8  9
8 char* itoa_var_length(std::uint32_t n, char* buffer) {
9     if (n < 100) {
10         if (n < 10) {
11             // 1 digit.
12             buffer[0] = char('0' + n);
13             return buffer + 1;
14         }
15         else {
16             // 2 digits.
17             std::memcpy(buffer, radix_100_table + n * 2, 2);
18             return buffer + 2;
19         }
20     }
21     else if (n < 100'0000) {
22         if (n < 1'0000) {
23             // 3 or 4 digits.
24             // 42949673 = ceil(2^32 / 10^2)
25             auto y = n * std::uint64_t(42949673);
26             if (n < 1'000) {
27                 // 3 digits.
28                 buffer[0] = char('0' + int(y >> 32));
29                 y = std::uint32_t(y) * std::uint64_t(100);
30                 std::memcpy(buffer + 1, radix_100_table + int(y >> 32) * 2, 2);
31                 return buffer + 3;
32             }
33             else {
34                 // 4 digits.
35                 std::memcpy(buffer + 0, radix_100_table + int(y >> 32) * 2, 2);
36                 y = std::uint32_t(y) * std::uint64_t(100);
37                 std::memcpy(buffer + 2, radix_100_table + int(y >> 32) * 2, 2);
38                 return buffer + 4;

```

```

39         }
40     }
41     else {
42         // 5 or 6 digits.
43         // 429497 = ceil(2^32 / 10^4)
44         auto y = n * std::uint64_t(429497);
45         if (n < 10'0000) {
46             // 5 digits.
47             buffer[0] = char('0' + int(y >> 32));
48             y = std::uint32_t(y) * std::uint64_t(100);
49             std::memcpy(buffer + 1, radix_100_table + int(y >> 32) * 2, 2);
50             y = std::uint32_t(y) * std::uint64_t(100);
51             std::memcpy(buffer + 3, radix_100_table + int(y >> 32) * 2, 2);
52             return buffer + 5;
53         }
54         else {
55             // 6 digits.
56             std::memcpy(buffer + 0, radix_100_table + int(y >> 32) * 2, 2);
57             y = std::uint32_t(y) * std::uint64_t(100);
58             std::memcpy(buffer + 2, radix_100_table + int(y >> 32) * 2, 2);
59             y = std::uint32_t(y) * std::uint64_t(100);
60             std::memcpy(buffer + 4, radix_100_table + int(y >> 32) * 2, 2);
61             return buffer + 6;
62         }
63     }
64 }
65 else if (n < 1'0000'0000) {
66     // 7 or 8 digits.
67     // 140737489 = ceil(2^47 / 10^6)
68     auto y = n * std::uint64_t(140737489);
69     constexpr auto mask = (std::uint64_t(1) << 47) - 1;
70     if (n < 1000'0000) {
71         // 7 digits.
72         buffer[0] = char('0' + int(y >> 47));
73         y = (y & mask) * 100;
74         std::memcpy(buffer + 1, radix_100_table + int(y >> 47) * 2, 2);
75         y = (y & mask) * 100;
76         std::memcpy(buffer + 3, radix_100_table + int(y >> 47) * 2, 2);
77         y = (y & mask) * 100;
78         std::memcpy(buffer + 5, radix_100_table + int(y >> 47) * 2, 2);
79         return buffer + 7;
80     }

```

```

81     else {
82         // 8 digits.
83         std::memcpy(buffer + 0, radix_100_table + int(y >> 47) * 2, 2);
84         y = (y & mask) * 100;
85         std::memcpy(buffer + 2, radix_100_table + int(y >> 47) * 2, 2);
86         y = (y & mask) * 100;
87         std::memcpy(buffer + 4, radix_100_table + int(y >> 47) * 2, 2);
88         y = (y & mask) * 100;
89         std::memcpy(buffer + 6, radix_100_table + int(y >> 47) * 2, 2);
90         return buffer + 8;
91     }
92 }
93 else {
94     // 9 or 10 digits.
95     // 1441151881 = ceil(2^57 / 10^8)
96     constexpr auto mask = (std::uint64_t(1) << 57) - 1;
97     auto y = n * std::uint64_t(1441151881);
98     if (n < 10'0000'0000) {
99         // 9 digits.
100        buffer[0] = char('0' + int(y >> 57));
101        y = (y & mask) * 100;
102        std::memcpy(buffer + 1, radix_100_table + int(y >> 57) * 2, 2);
103        y = (y & mask) * 100;
104        std::memcpy(buffer + 3, radix_100_table + int(y >> 57) * 2, 2);
105        y = (y & mask) * 100;
106        std::memcpy(buffer + 5, radix_100_table + int(y >> 57) * 2, 2);
107        y = (y & mask) * 100;
108        std::memcpy(buffer + 7, radix_100_table + int(y >> 57) * 2, 2);
109        return buffer + 9;
110    }
111 }
112 else {
113     // 10 digits.
114     std::memcpy(buffer + 0, radix_100_table + int(y >> 57) * 2, 2);
115     y = (y & mask) * 100;
116     std::memcpy(buffer + 2, radix_100_table + int(y >> 57) * 2, 2);
117     y = (y & mask) * 100;
118     std::memcpy(buffer + 4, radix_100_table + int(y >> 57) * 2, 2);
119     y = (y & mask) * 100;
120     std::memcpy(buffer + 6, radix_100_table + int(y >> 57) * 2, 2);
121     y = (y & mask) * 100;
122     std::memcpy(buffer + 8, radix_100_table + int(y >> 57) * 2, 2);

```

```

123     return buffer + 10;
124 }
125 }
126 }
```

Note: The paths for $(2k-1)$ -digits case and $2k$ -digits case share a lot of code, so one might try to merge the code for printing $(2k-2)$ remaining digits while leaving in separate branches only the code for printing the leading 1 or 2 digits. However, it seems that such a refactoring causes the code to perform worse, probably because the number of additions performed is increased. Nevertheless, that is also one viable option, especially regarding the code size.

57.6 Better choices for y

The above code is pretty good for n' s up to 6 digits, but not so much for longer n' s, as we have to perform masking in addition to multiplication and shift for each 2 digits. This is due to D being not equal to 32, so it will be beneficial if we can choose $D = 32$ even for n' s with digits more than 6. And it turns out that we can.

The reason we had to choose $D > 32$ was due to our poor choice of y :

$$y = n \left\lceil \frac{2^D}{10^k} \right\rceil \quad (57.18)$$

Recall that we do not need to choose y like this; all we need to ensure is that y satisfies the inequality

$$\frac{2^D n}{10^k} \leq y < \frac{2^D (n+1)}{10^k} \quad (*)$$

Slightly generalizing what we have done, let us suppose that we want to obtain y by computing

$$y = \left\lceil \frac{nm}{2^L} \right\rceil \quad (57.19)$$

for some positive integer constants m and L . Then the inequality $(*)$ can be rewritten as

$$\frac{1}{n} \left\lceil \frac{2^D n}{10^k} \right\rceil \leq \frac{m}{2^L} < \frac{1}{n} \left\lceil \frac{2^D (n+1)}{10^k} \right\rceil. \quad (**)$$

At the time of writing this post, I am not quite sure if there is an elegant way to obtain the precise admissible range of m and L for the above inequality with any given range of n , but a reasonable guess is that

$$m = \left\lceil \frac{2^{D+L}}{10^k} \right\rceil + 1 \quad (57.20)$$

will often do the job. Indeed, in this case we have

$$\frac{mn}{2^L} \geq \frac{2^D n}{10^k} + \frac{n}{2^L}, \quad (57.21)$$

so the left-hand side of (**) is always satisfied if

$$2^L \leq n \quad (57.22)$$

holds for all n in the range. On the other hand, we have

$$\frac{mn}{2^L} < \frac{2^D n}{10^k} + \frac{n}{2^{L-1}}, \quad (57.23)$$

so the right-hand side of (**) is always satisfied if

$$\frac{n}{2^{L-1}} \leq \frac{2^D}{10^k}, \quad (57.24)$$

or equivalently,

$$\frac{10^k n}{2^{D-1}} \leq 2^L \quad (57.25)$$

holds for all n in the range.

Thus for example, when n is of 7 or 8 digits (so $n \in [10^6, 10^8-1]$), $k = 6$, and $D = 32$, it is enough to have

$$\frac{10^6 (10^8 - 1)}{2^{31}} \leq 2^L \leq 10^6, \quad (57.26)$$

which is equivalent to

$$16 \leq L \leq 19. \quad (57.27)$$

Hence, we take $L = 16$ and accordingly $m = 281474978$. Of course, since m is even we can equivalently take $L = 15$ and $m = 140737489$ as well, so this method of analysis is clearly far from being tight.

(In fact, it can be exhaustively verified that the left-hand side of (*) is maximized when $n = 1000795$, while the right-hand side is minimized when $n = 10^8 - 1$, which together yield the inequality

$$\frac{4298381796}{1000795} \leq \frac{m}{2^L} < \frac{429496729600}{99999999} \quad (57.28)$$

The minimum L allowing an integer solution for m to the above inequality is $L = 15$ and in this case $m = 140737489$ is the unique solution.)

When n is of 9 or 10 digits, a similar analysis does not give the best result, especially for 10 digits it fails to give any admissible choice of L and m . Nevertheless, it can be exhaustively verified that, when $k = 8$ and $D = 32$, if we set $L = 25$ and

$$m = \left\lceil \frac{2^{D+L}}{10^k} \right\rceil + 1 = 1441151882, \quad (57.29)$$

then

$$n = \left\lceil \frac{10^k \lfloor nm/2^L \rfloor}{2^D} \right\rceil \quad (57.30)$$

holds for all $n \in [10^8, 10^9 - 1]$, and similarly, if we set $L = 25$ and

$$m = \left\lceil \frac{2^{D+L}}{10^k} \right\rceil = 1441151881, \quad (57.31)$$

then

$$n = \left\lceil \frac{10^k \lfloor nm/2^L \rfloor}{2^D} \right\rceil \quad (57.32)$$

holds for all $n \in [10^9, 2^{32} - 1]$, so we can still take

$$y = \left\lfloor \frac{nm}{2^L} \right\rfloor \quad (57.33)$$

with the above L and m .

(In fact, applying what we have done for 7 or 8 digits into the case of 9 digits gives $L = 26$ and $m = 2882303763$. However, 1441151882 is a better magic number than 2882303763, because the former is of 31-bits while the latter is of 32-bits. This trivially-looking difference actually quite matters on platforms like x86, because when computing $y = \lfloor \frac{nm}{2^L} \rfloor$, we want to leverage the fast `imul` instruction, but `imul` sign-extends the input immediate constant when performing 64-bit multiplication. Hence, if the magic number is of 32-bits, the multiplication cannot be done in a single instruction, and the magic number must be first loaded into a register and then zero-extended.)

Therefore, we are indeed able to always choose $D = 32$, which results in the following code:

```

1 // https://godbolt.org/z/7TaqYa9h1
2 char* itoa_better_y(std::uint32_t n, char* buffer) {
3     std::uint64_t prod;
4
5     auto get_next_two_digits = [&]() {
6         prod = std::uint32_t(prod) * std::uint64_t(100);
7         return int(prod >> 32);
8     };
9     auto print_1 = [&](int digit) {
10         buffer[0] = char(digit + '0');
11         buffer += 1;
12     };
13     auto print_2 = [&] (int two_digits) {
14         std::memcpy(buffer, radix_100_table + two_digits * 2, 2);
15         buffer += 2;
16     };
17     auto print = [&](std::uint64_t magic_number, int extra_shift, auto remaining_count) {
18         prod = n * magic_number;

```

```

19     prod >>= extra_shift;
20     auto two_digits = int(prod >> 32);
21
22     if (two_digits < 10) {
23         print_1(two_digits);
24         for (int i = 0; i < remaining_count; ++i) {
25             print_2(get_next_two_digits());
26         }
27     }
28     else {
29         print_2(two_digits);
30         for (int i = 0; i < remaining_count; ++i) {
31             print_2(get_next_two_digits());
32         }
33     }
34 };
35
36 if (n < 100) {
37     if (n < 10) {
38         // 1 digit.
39         print_1(n);
40     }
41     else {
42         // 2 digit.
43         print_2(n);
44     }
45 }
46 else {
47     if (n < 100'0000) {
48         if (n < 1'0000) {
49             // 3 or 4 digits.
50             // 42949673 = ceil(2^32 / 10^2)
51             print(42949673, 0, std::integral_constant<int, 1>{});
52         }
53         else {
54             // 5 or 6 digits.
55             // 429497 = ceil(2^32 / 10^4)
56             print(429497, 0, std::integral_constant<int, 2>{});
57         }
58     }
59     else {
60         if (n < 1'0000'0000) {

```

```

61     // 7 or 8 digits.
62     // 281474978 = ceil(2^48 / 10^6) + 1
63     print(281474978, 16, std::integral_constant<int, 3>{});
64 }
65 else {
66     if (n < 10'0000'0000) {
67         // 9 digits.
68         // 1441151882 = ceil(2^57 / 10^8) + 1
69         prod = n * std::uint64_t(1441151882);
70         prod >>= 25;
71         print_1(int(prod >> 32));
72         print_2(get_next_two_digits());
73         print_2(get_next_two_digits());
74         print_2(get_next_two_digits());
75         print_2(get_next_two_digits());
76     }
77     else {
78         // 10 digits.
79         // 1441151881 = ceil(2^57 / 10^8)
80         prod = n * std::uint64_t(1441151881);
81         prod >>= 25;
82         print_2(int(prod >> 32));
83         print_2(get_next_two_digits());
84         print_2(get_next_two_digits());
85         print_2(get_next_two_digits());
86         print_2(get_next_two_digits());
87     }
88 }
89 }
90 }
91 return buffer;
92 }
```

Note

Looking at a port^a of James Anhalt's original algorithm, it seems that the above code is probably a little bit better than the original implementation (which can be confirmed in the benchmark below) because the original algorithm performs an addition after the first multiplication and shift, for digit length longer than some value. With our choice of magic numbers, that is not necessary.

^ahttps://github.com/tearosccebe/fast_io/blob/e74bd525b6765a9f418137d9aebd193f133e400e/include/fast_io_core_impl/integers/jcaiii_method.h#L49

57.7 Benchmark

Alright, now let's compare the performance of these implementations. (fig. 57.1)

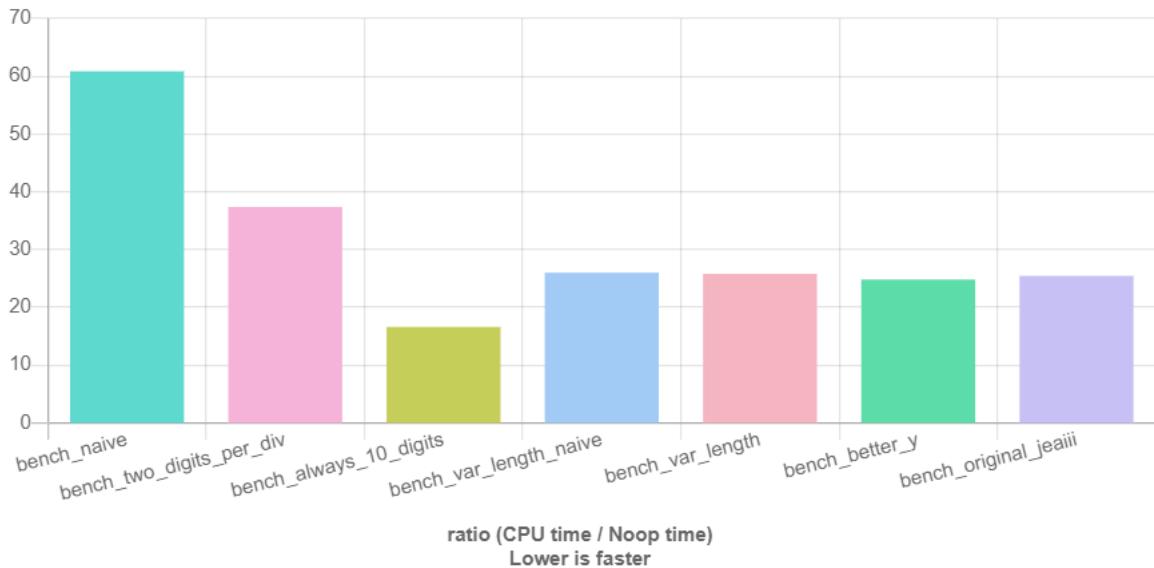


图 57.1: <https://quick-bench.com/q/ieIpsBhWC751YUhyVS2OE83dTO4>

`itoa_var_length_naive` is a straightforward variation of `itoa_var_length` doing the naive quotient/remainder computation instead of playing with y . Well, compared to `itoa_var_length_naive`, the performance benefit of `itoa_better_y` seems not very impressive to be honest. Nevertheless, I still think the idea behind the algorithm is pretty intriguing.

57.8 Back to fixed-length case

So far we only have looked at the case of 32-bit unsigned integers. For 64-bit integers, what people typically do is to first divide the input number into several segments of 32-bit integers, and then print them using methods for 32-bit integers. In this case, typically only the first segment is of variable length and remaining segments are of fixed length. As we can see in the benchmark above, when the length is known we can do a lot better. For simplicity of the following discussion, let us suppose that the input integer is at most of 9 digits and we want to always print 9 digits with possible leading zeros.

While what we have done in `itoa_always_10_digits` is not so bad, we can certainly do better by choosing $D = 32$ which eliminates the need for performing masking at each generation of 2 digits. Recall that all we need to do is to find an integer y satisfying

$$\frac{2^D n}{10^k} \leq y < \frac{2^D (n + 1)}{10^k} \quad (*)$$

for given n . Since we want to always print 9 digits, we take $k = 8$. What's different from the previous case is that now n can be any integer in the range $[1, 10^9 - 1]$ (ignoring $n = 0$ case, but that is not a big deal), in particular it can be very small. In this case, one can show by exhaustively checking all possible n 's that

the inequality

$$\frac{1}{n} \left\lceil \frac{2^D n}{10^k} \right\rceil \leq \frac{m}{2^L} \leq \frac{1}{n} \left\lceil \frac{2^D (n+1)}{10^k} \right\rceil \quad (**)$$

does not have a solution, because the maximum value of the left-hand side is bigger than the minimum value of the right-hand side. Therefore, it is not possible to compute y by just performing a multiplication followed by a shift.

Instead, we choose

$$y = \left\lfloor \frac{nm}{2^L} \right\rfloor + 1, \quad (57.34)$$

which means that we indeed omit masking at each generation of 2 digits, but at the cost of additionally performing an increment for the initial step of computing y .

With this choice of y , $(*)$ becomes

$$\frac{1}{n} \left\lceil \frac{2^D n}{10^k} \right\rceil - \frac{1}{n} \leq \frac{m}{2^L} \leq \frac{1}{n} \left\lceil \frac{2^D (n+1)}{10^k} \right\rceil - \frac{1}{n} \quad (**')$$

instead of $(**)$. Then we can perform a similar analysis to conclude that $L = 25$ with

$$m = \left\lceil \frac{2^{D+L}}{10^k} \right\rceil = 1441151881 \quad (57.35)$$

do the job. In fact, an exhaustive check shows that we can even take $L = 24$ and $m = 720575941$.

EDIT: Actually, $L = 24$ with $m = 720575941$ doesn't work for large numbers. See here⁴ for a better analysis.

57.9 Concluding remarks

I applied a minor variation of the algorithm explained here into my Dragonbox implementation⁵ to speed up digit generation, and the result was quite satisfactory. I think probably the complete branching for all possible lengths of the input is not a brilliant idea for generic applications, but the idea of coming up with y that enables generating each pair of 2 digits by only performing a multiply-and-shift is very clever and useful. I expect that this same trick might be applicable to other problems as well, fixed-precision floating-point formatting for example.

⁴<https://jk-jeon.github.io/posts/2022/12/fixed-precision-formatting/#appendix-fixed-point-fraction-trick-revisited>

⁵https://github.com/jk-jeon/dragonbox/blob/master/source/dragonbox_to_chars.cpp

Part VIII

Techniques

本 Part 所含技术丰富多彩，内容偏底层和通用。

简要概述如下：

- **Chapter 58:** 关于如何实现条件成员的技术。其采用`[[no_unique_address]]` 和`std::conditional_t` 实现类的条件成员变量，并使用`concept` 实现条件成员函数，通过继承实现条件成员类型；
- **Chapter 59,60:** 关于`boost::unordered_flat_map` 和静态B-tree 数据结构的实现细节剖析，难度略高，对标准库实现细节感兴趣的可以一观；
- **Chapter 61,64:** 关于`std::variant` 的两个特殊应用，难度适中；
- **Chapter 62:** 详细对比了标准库中三种字符串类型的应用，不局限于嵌入式，难度较低，属入门级材料；
- **Chapter 63:** 介绍了一种基于`lambda` 封装的技术，从而在编译期检测函数模板特化是否存在，思路清晰简洁；
- **Chapter 65:** 介绍了一种基于宏的对象序列化方法，通过可变参数宏实现遍历操作；
- **Chapter 66:** 讲解了如何将`constexpr string` 的值保存到运行期使用的技术。

通过本部分内容，相信大家能够学到不少新东西。

2023年5月15日
水滴会飞

Chapter 58

Conditional Members

• Barry Revzin  2021-11-21  

I'd previously written a post about `if constexpr` (and how it's not broken¹). I argued in that post how, broadly speaking, C++20 gives you the tools to solve the problems you want, even if they work a bit differently to D's `static if` (with one notable exception, which this post greatly expands on). Now, over the past couple years, I've been working on a project that really is a deep dive into what Andrei calls "Design by Introspection." This approach (for lack of a better definition), relies on conditioning functionality based on template parameters.

For the purposes of this post, I'm going to deal with one particular kind of design by introspection: having conditional members. There are three kinds of members that we want to be able to have conditionally (or differently) present based on template parameters:

- conditional member *functions*
- conditional member *variables*
- conditional member *types*

These are arranged roughly in how easy it is to do them in C++20, and I'll go through the issues that come up with each of these in turn. In D, you basically have one language feature to solve all of these problems and that one language feature is `if` (sometimes `static if`). In C++, we use different tools in each case.

58.1 Conditional Member Functions

The goal here is to create a member function that only exists when the template parameters meet some criteria.

In C++20, Concepts are basically the way that we solve this problem. In C++17 and earlier, we could always add "constraints" to member function templates of class templates (i.e. using `std::enable_if`), but we couldn't add them to member functions that were not templates. The most important of these are the special member functions. You can't make a class conditionally copyable with `std::enable_if`. But in C++20, you can add proper constraints (no scare quotes necessary) in all of these cases. And that just works:

¹<https://brevzin.github.io/c++/2019/01/15/if-constexpr-isnt-broken/>

```
template <typename T>
class Optional {
public:
    Optional(Optional const&) requires copy_constructible<T>;
};
```

or:

```
template <input_range R>
class adapted_range {
public:
    constexpr auto size() requires sized_range<R>;
};
```

In these examples, `Optional<int>` would be copy constructible, but `Optional<unique_ptr<int>>` wouldn't be. `adapted_range<vector<int>>` would have a `size()` member function but `adapted_range<filter_view<V, F>>` would not.

Using C++20 concepts to conditionally control member functions just works great.

Nearly all the time. There's one kind of exception. Consider this case of writing a smart pointer. Being a pointer, I want to provide an `operator*`. But, because I also support `void`, and you can't dereference a `void*`, I need to make sure that this member function does not exist in that case. Naturally, I'll use concepts:

```
template <typename T>
struct Ptr {
    auto operator*() const -> T&
        requires (!std::is_void_v<T>);
};
```

```
Ptr<void> p; // error
```

That is already ill-formed². I'm not even trying to `*p` anywhere, simply creating the type. What happened to my constraint?

The issue here is that Concepts don't actually do conditional member functions. It's not that `Optional<unique_ptr<int>>` had no copy constructor or that `adapted_range<filter_view<V, F>>` had no member function named `size()`. They do have those functions. It's just that, when it comes to overload resolution, those (actually-existing) functions are removed from consideration at that point.

Typically, there's no distinction between these cases. There's not really much of a difference between `adapted_range` not having a `size()` member function and it having one that you simply cannot invoke. You can't really differentiate.

But in this case there is.

The rule is that when a class template is instantiated (as in the declaration of `Ptr<void> p` above), all of the signatures of its member functions are instantiated (this is [temp.inst]/3³). And doing so requires forming `T&`, which is not a valid thing to do when `T` is `void`, and this blows up at that point (gcc and clang's errors clearly point to this, MSVC's not so much).

²<https://godbolt.org/z/4TbMsfxhc>

³<http://eel.is/c++draft/temp.inst#3>

The way to do this correctly in C++20 is to either wrap the `T&` in something that correctly handles `void` (`std::add_lvalue_reference_t<void>` is `void`):

```
template <typename T>
struct Ptr {
    auto operator*() const -> std::add_lvalue_reference_t<T>
        requires (!std::is_void_v<T>);
};

Ptr<void> p; // ok
```

Or to turn the whole function into a function template to delay its instantiation (now we’re returning `U&`, not `T&`):

```
template <typename T>
struct Ptr {
    template <typename U=T> requires (!std::is_void_v<U>)
        auto operator*() const -> U&;
};

Ptr<void> p; // also ok
```

Personally though, I dislike both of these solutions. The goal here is to have `operator*` exist only when `T` isn’t `void`. Concepts unfortunately don’t help here.

But Concepts do help most of the rest of the time. While they don’t literally give us conditional member functions, they do basically help solve that problem. Conditional Member Variables

If you browse through the spec for Ranges ([ranges]⁴), there are several cases where we want to have a member variable that is present only under certain conditions. Ranges isn’t that unique in this sense, there are plenty of situations where this sort of thing comes up.

In the Standard, we write (I’m omitting some of the template parameters here for brevity and clarity):

```
template <input_range V>
struct lazy_split_view<V>::outer_iterator {
    // exposition only, present only if !forward_range<V>
    iterator_t<V> current_ = iterator_t<V>();
};
```

But we have to write it as something like this:

```
template <input_range V>
struct lazy_split_view<V>::outer_iterator {
    struct empty { };
    using current_t = std::conditional_t<
        !forward_range<V>, iterator_t<V>, empty>;
    [[no_unique_address]] current_t current_ = current_t();
};
```

⁴<http://eel.is/c++draft/ranges>

Just like we saw in the previous section, and perhaps more obviously here, this isn't *really* a conditional member. `current_`, as a member, is always present. It's just that we can concoct a solution that avoids space overhead thanks to `[[no_unique_address]]`.

If we're especially paranoid, we can help ensure that all of these empty types are distinct by taking advantage of lambdas:

```
namespace N {
    template <typename T> struct empty_type {
        // add a constructor from anything to make conditional
        // initialization easier to deal with
        constexpr empty_type(auto&&...) { }
    };
}

#define EMPTY_TYPE ::N::empty_type<decltype([]{})>
```

Now, every use of `EMPTY_TYPE` is a distinct type. Which is fine, because the only time we'd use such a thing is here:

```
template <input_range V>
struct lazy_split_view<V>::outer_iterator {
    using current_t = std::conditional_t<
        !forward_range<V>, iterator_t<V>, EMPTY_TYPE>;
    [[no_unique_address]] current_t current_ = current_t();
};
```

But `current_` is still *always* present. It's not really a conditional member, only its type is conditional. In my experience with needing conditional members at least, having an empty placeholder has been thankfully sufficient.

But there is a case where we really need the member to be truly conditional, and that is...

58.2 Conditional Member Types

The most familiar example of needing a conditional type in C++ is one that I've already hinted at earlier: `std::enable_if`. `enable_if` is nothing more than wanting a type that's either there, or not. If we were specifying it Ranges-style, we'd write it this way:

```
template <bool B, typename T>
struct enable_if {
    using type = T; // present only if B is true
};
```

Here, it's critical that `enable_if<false, T>` has no member type at all. Not has a member type that is `void` or some other implementation-defined type. No type at all!

There is only one way to do this in the language today, which is partial specialization of class templates. You have to do it this way:

```
template <bool B, typename T>
struct enable_if {
    using type = T;
};

template <typename T>
struct enable_if<false, T> { };
```

Or the reverse - have the partial specialization handle the `true` case. Either way.

For a utility like `enable_if`, writing this across multiple specializations isn't that big a deal. Mildly tedious at best. But once you start writing bigger utilities, this becomes a lot more than mildly tedious. Suddenly you have to come up with a crazy workaround.

One of the many new range adaptors in C++23 will be `zip_transform`. Some languages call this `zip_with`: this is a `zip` that additionally takes a function that gets applied to each corresponding argument in all the ranges. For example:

```
vector v1 = {1, 2};
vector v2 = {4, 5, 6};

fmt::print("{}\n", views::zip_transform(plus(), v1, v2)); // [5, 7]
```

Now, if you look at the specification for the iterator (in [range.zip.transform.iterator]⁵), you'll see that its `iterator` has a member type, `iterator_category`, that is only conditionally present:

```
template <copy_constructible F, input_range... Views>
    requires /* ... */
template <bool Const>
class zip_transform_view<F, Views...>::iterator {
    // ...
public:
    using iterator_category = see below; // not always present
    // ...
}
```

The definition of `iterator_category` is complicated (see [range.zip.transform.iterator]/1⁶). But importantly it's only present if all the underlying ranges are forward ranges. And then, when it is present, it's just basically the common category of all the underlying ranges. The wording is a bit involved here, but the underlying operation isn't that complex.

So...how do you do that?

We need the same kind of logic as with `enable_if`, we need a partial specialization. We can start by thinking of it this way:

```
template <bool B, typename T>
struct maybe_iterator_category {
```

⁵<http://eel.is/c++draft/range.zip.transform.iterator>

⁶<http://eel.is/c++draft/range.zip.transform.iterator#1>

```

    using iterator_category = T;
};

template <typename T>
struct maybe_iterator_category<false, T> { };


```

However, the nuance here is that you cannot check for the underlying ranges' `iterator_category` types until we very that they even have them - which means we have to delay evaluation of those traits. The way I implemented this in my approach to `zip_transform` when Tim Song was working on the paper (P2321⁷), was to instead take a page out of Boost.Mp11⁸'s book:

```

template <bool B, template <typename...> class F, typename... T>
struct maybe_iterator_category {
    using iterator_category = F<T...>;
};

template <typename T>
struct maybe_iterator_category<false, T> { };


```

Which I used like so⁹ (this is on lines 713-729):

```

1 template <typename T>
2 using nested_iterator_category = typename T::iterator_category;
3
4 template <typename I>
5 using iterator_category_for = mp_eval_or<
6     std::input_iterator_tag, nested_iterator_category, iterator_traits<I>>;
7
8 template <bool Const>
9 using categories = mp_list<
10    iterator_category_for<iterator_t<maybe_const<Const, Views>>>...>;
11
12 template <bool Const, typename Tag>
13 using all_categories_derive_from = mp_all_of_q<
14    categories<Const>, mp_bind_front<is_base_of, Tag>>;
15
16 template <bool Const>
17 using result_type = invoke_result_t<
18    maybe_const<Const, F>&, range_reference_t<maybe_const<Const, Views>>...>;
19
20 template <bool Const>
21 class iterator : public maybe_iterator_category<


```

⁷<https://wg21.link/p2321>

⁸<https://www.boost.org/doc/libs/develop/libs/mp11/doc/html/mp11.html>

⁹<https://godbolt.org/z/df7dTj4nM>

```

22     // only present if Base models forward_range
23     forward_range<maybe_const<Const, InnerView>>,
24     mp_cond,
25     mp_bool<!is_lvalue_reference_v<result_type<Const>>>,
26     input_iterator_tag,
27     all_categories_derive_from<Const, random_access_iterator_tag>,
28     random_access_iterator_tag,
29     all_categories_derive_from<Const, bidirectional_iterator_tag>,
30     bidirectional_iterator_tag,
31     all_categories_derive_from<Const, forward_iterator_tag>,
32     forward_iterator_tag,
33     mp_true,
34     input_iterator_tag>
35 {
36     // ...
37 };

```

There are other slightly different approaches, but they all basically have to jump through the same hoops. You have to inherit from something in order to properly have a conditional member `iterator_category`. You need to come up with some way to delay checking `T::iterator_category` until after we know that we're a `forward_range` (I chose to do this by instead using `std::input_iterator_tag` as the default if there is no `iterator_category` - this default will never be used, but it allows me to write all the conditions in-line, although it makes all the conditions more complex).

But this is basically the only way to really have a true *conditional* member in C++: you have to inherit from either a type that has that member or a type that does not have that member. I didn't show this in the original example with `Ptr` as an alternative implementation, you could do that too:

```

template <typename T>
struct PtrBase { };

template <typename T> requires (!std::is_void_v<T>)
struct PtrBase {
    auto operator*() const -> T&;
};

template <typename T>
struct Ptr : PtrBase<T>
{ };

Ptr<void> p; // ok

```

And the reason I didn't show this, or indeed consider this a real viable alternative, is that it's...a pretty bad alternative. It's very verbose and disruptive to comprehension (and this even without dealing with how do you implement `PtrBase`'s `operator*?`). In the `Ptr<T>` case, I didn't actually need `operator*`

`or*` to be truly absent, I just needed to delay instantiation of `operator*`. But in the `zip_transform<F, V...>::iterator` case, I do actually *need iterator_category* to be truly absent.

So inheriting from a conditional base class it is.

It's worth noting here that inheriting from a conditional base class is how we used to have to do conditional member variables as well, if you wanted to avoid the storage overhead. If you look at the paper that gave us `[[no_unique_address]]` (P0840R0¹⁰), the tool that lets us avoid the conditional base shenanigans when we're dealing with conditional member variables, the paper clearly points out one huge downside of the original approach:

Implementation awkwardness: [Empty Base Optimization] requires state that would naturally be represented as a data member to be moved into a base class.

Awkwardness indeed.

58.3 An alternate approach

If we look back on these problems, we used three different language features to handle them:

- conditional member variable: `[[no_unique_address]]` with `std::conditional_t` (and if we're really insistent, unevaluated lambdas).
- conditional member function: concepts.
- conditional member type: inheriting from conditional base classes.

Of these, only the last option actually handles all the cases, and only the last option gives you truly a conditional member. It's also the most inconvenient/awkward/tedious/insert pejorative of your choice.

However, D gives us what I think is a clear answer for how we could do conditional members in a way that is properly conditional and avoids the kind of tedium that we have to deal with today: `if`.

We could just use `if` at class scope to declare a conditional member function (no need to come up with a workaround to wrap `T&`):

```
template <typename T>
struct Ptr {
    if (!std::is_void_v<T>) {
        auto operator*() const -> T&
    }
};
```

We could just use `if` at class scope to declare a conditional member variable (no need to come up with a workaround for how to declare the type of `current_` such that it's empty, we can directly use the type that we want for the member everywhere - including its initializer):

```
template <input_range V>
struct lazy_split_view<V>::outer_iterator {
    if (!forward_range<V>) {
```

¹⁰<https://wg21.link/p0840r0>

```

    iterator_t<V> current_ = iterator_t<V>();
}
};


```

And, most significantly, we could just use `if` at class scope to declare a conditional member type:

```

template <typename F, typename... Vs>
struct zip_transform_view<F, Vs...>::iterator {
    if ((forward_range<Vs> && ...)) {
        if /* not a reference */ {
            using iterator_category = std::input_iterator_tag;
        } else {
            // here we can eagerly access all of the iterator_category's because
            // we know that they exist (because of forward_range)
            using iterator_category = std::common_type_t<
                std::random_access_iterator_tag,
                typename iterator_traits<iterator_t<Vs>::iterator_category...
            >;
        }
    }
};


```

Now here we of course run into the scope problem. `if` introduces a scope, so all of these code fragments look very much like they’re introducing something which only exist in the scope in which it’s declared (which would then be, at best, a completely pointless exercise). It’d be important to work through the rules of what it actually means to introduce these names and members in these contexts, which will, I’m sure, be subtle and full of dark corners. And while I think that here, a scope-less `if` would be valuable, I still don’t feel that `if constexpr` is missing much for introducing a scope (indeed, quite the opposite).

The direction for reflection (P2237¹¹, P2320¹²) does offer something like this. The syntax is a work in flight, but would replace this example I just showed:

```

template <input_range V>
struct lazy_split_view<V>::outer_iterator {
    if (!forward_range<V>) {
        iterator_t<V> current_ = iterator_t<V>();
    }
};


```

with something like this (at some point I think the injection operator changed from `<<` to `<-` but I can’t find that in the paper, and in any case the specific syntax here is less important than the overall shape of the solution, which I think is about right):

```

template <input_range V>
struct lazy_split_view<V>::outer_iterator {


```

¹¹<https://wg21.link/p2237>

¹²<https://wg21.link/p2320>

```

consteval {
    if (!forward_range<V>) {
        << struct { iterator_t<V> current_ = iterator_t<V>(); }>;
    }
}
};

```

There are a few new things that are new here: a **consteval** block, a code fragment (the **<struct ...>** part), and an injection statement. And this allows for clear definitions of when things happening (in particular, at the end of a **consteval** block, all the fragments queued for injection are actually injected). There's certainly value in having a clear model for things, especially in an area with as much subtlety as this.

I want to be clear that the reason I dislike the **consteval** block approach is not because it's more verbose. It is, but not by a lot. And certainly if I had a choice between the latter and nothing I would choose the latter in an instant. We often talk about verbosity, but I think terseness is only especially important in a few key circumstances (like lambdas¹³) - and oftentimes terseness is the wrong goal and can significantly harm readability and adoption (c.f. build2). Here the problem isn't strictly that the **consteval** block approach is longer - the problem for me is that none of the additional syntax actually adds meaning on top of the shorter version that's just the **if** statement. The **if** approach isn't just terser for the sake of terseness, and I didn't get there by introducing some grawlix punctuation. It's just the same kind of **if** that we're already familiar with - just in a different context.

As a result I'm hard-pressed to see why we can't just...make the former example mean the latter example. Which would allow us to just have conditional members the same way we write all of our other conditions: with **if**. This isn't to say the **consteval** block approach isn't useful, it certainly is (such as wanting to write a function that returns a code fragment, and inject that - you need some kind of thing to be able to return from such a function, and this is important). Just that the simple case probably merits avoiding some of the ceremony.

Regardless, we do need a better way to express conditional members than what we have today. This isn't that rare a problem, and currently we take very different approaches based on the kind of member we're conditioning, each of which has different nuances and issues.

¹³<https://brevzin.github.io/c++/2020/06/18/lambda-lambda-lambda/>

Chapter 59

Inside boost::unordered_flat_map

👤 Joaquín M López Muñoz 📅 2022-09-18 💬 ★★★★

59.1 Introduction

Starting in Boost 1.81 (December 2022), Boost.Unordered¹ provides, in addition to its previous implementations of C++ unordered associative containers, the new containers `boost::unordered_flat_map` and `boost::unordered_flat_set` (for the sake of brevity, we will only refer to the former in the remaining of this article). If `boost::unordered_map` strictly adheres to the C++ specification for `std::unordered_map`, `boost::unordered_flat_map` deviates in a number of ways from the standard to offer dramatic performance improvements in exchange; in fact, `boost::unordered_flat_map` ranks amongst the fastest hash containers currently available to C++ users.

We describe the internal structure of `boost::unordered_flat_map` and provide theoretical analyses and benchmarking data to help readers gain insights into the key design elements behind this container's excellent performance. Interface and behavioral differences with the standard are also discussed.

59.2 The case for open addressing

We have previously discussed² why *closed addressing* was chosen back in 2003 as the implicit layout for `std::unordered_map`. 20 years after, open addressing³ techniques have taken the lead in terms of performance, and the fastest hash containers in the market all rely on some variation of open addressing, even if that means that some deviations have to be introduced from the baseline interface of `std::unordered_map`.

The defining aspect of open addressing is that elements are stored directly within the bucket array (as opposed to closed addressing, where multiple elements can be held into the same bucket, usually by means of a linked list of nodes). In modern CPU architectures, this layout is extremely cache friendly:

- There's no indirection needed to go from the bucket position to the element contained.
- Buckets are stored contiguously in memory, which improves cache locality.

¹<https://www.boost.org/doc/libs/release/libs/unordered/>

²<https://bannalia.blogspot.com/2022/06/advancing-state-of-art-for.html>

³https://en.wikipedia.org/wiki/Hash_table#Open_addressing

The main technical challenge introduced by open addressing is what to do when elements are mapped into the same bucket, i.e. when a *collision* happens: in fact, all open-addressing variations are basically characterized by their collision management techniques. We can divide these techniques into two broad classes:

- **Non-relocating:** if an element is mapped to an occupied bucket, a *probing sequence* is started from that position until a vacant bucket is located, and the element is inserted there *permanently* (except, of course, if the element is deleted or if the bucket array is grown and elements *rehashed*). Popular probing mechanisms are *linear probing* (buckets inspected at regular intervals), *quadratic probing* and *double hashing*⁴. There is a tradeoff between cache locality, which is better when the buckets probed are close to each other, and *average probe length* (the expected number of buckets probed until a vacant one is located), which grows larger (worse) precisely when probed buckets are close —elements tend to form clusters instead of spreading uniformly throughout the bucket array.
- **Relocating:** as part of the search process for a vacant bucket, elements can be moved from their position to make room for the new element. This is done in order to improve cache locality by keeping elements close to their "natural" location (that indicated by the hash → bucket mapping). Well known relocating algorithms are *cuckoo hashing*⁵, *hopscotch hashing*⁶ and *Robin Hood hashing*⁷.

If we take it as an important consideration to stay reasonably close to the original behavior of `std::unordered_map`, relocating techniques pose the problem that `insert` may invalidate iterators to other elements (so, they work more like `std::vector::insert`).

On the other hand, non-relocating open addressing faces issues on deletion: lookup starts at the original hash → bucket position and then keeps probing till the element is found *or probing terminates*, which is signalled by the presence of a vacant bucket:



图 59.1: probe

So, erasing an element can't just restore its holding bucket as vacant, since that would preclude lookup from reaching elements further down the probe sequence:



图 59.2: probe_interrupted

A common technique to deal with this problem is to label buckets previously containing an element with a tombstone marker: *tombstones* are good for inserting new elements but do not stop probing on lookup:

Note that the introduction of tombstones implies that the average lookup probe length of the container won't decrease on deletion —again, special measures can be taken to counter this.

⁴https://en.wikipedia.org/wiki/Double_hashing

⁵https://en.wikipedia.org/wiki/Cuckoo_hashing

⁶https://en.wikipedia.org/wiki/Hopscotch_hashing

⁷https://en.wikipedia.org/wiki/Hash_table#Robin_Hood_hashing

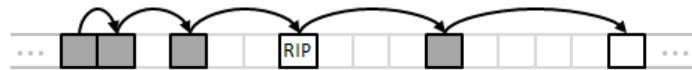


图 59.3: probe_tombstone

59.2.1 SIMD-accelerated lookup

SIMD technologies, such as SSE⁸ and Neon⁹, provide advanced CPU instructions for parallel arithmetic and logical operations on groups of contiguous data values: for instance, SSE2 `_mm_cmpeq_epi8`¹⁰ takes two packs of 16 bytes and compares them for equality *pointwise*, returning the result as another pack of bytes. Although SIMD was originally meant for acceleration of multimedia processing applications, the implementors of some unordered containers, notably Google's Abseil's Swiss tables¹¹ and Meta's F14¹², realized they could leverage this technology to improve lookup times in hash tables.

The key idea is to maintain, in addition to the bucket array itself, a separate *metadata array* holding *reduced hash values* (usually one byte in size) obtained from the hash values of the elements stored in the corresponding buckets. When looking up for an element, SIMD can be used on a pack of contiguous reduced hash values to quickly discard non-matching buckets and move on to full comparison for matching positions. This technique effectively checks a moderate number of buckets (16 for Abseil, 14 for F14) in constant time. Another beneficial effect of this approach is that special bucket markers (vacant, tombstone, etc.) can be moved to the metadata array —otherwise, these markers would take up extra space in the bucket itself, or else some representation values of the elements would have to be restricted from user code and reserved for marking purposes.

`boost::unordered_flat_map` data structure

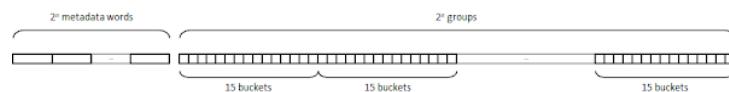


图 59.4: boost::unordered_flat_map data structure

`boost::unordered_flat_map`'s bucket array is logically split into 2^n groups of $N = 15$ buckets, and has a companion metadata array consisting of $2n$ 16-byte words. Hash mapping is done at the group level rather than on individual buckets: so, to insert an element with hash value h , the group at position $h/2^{W-n}$ is selected and its first available bucket used (W is 64 or 32 depending on whether the CPU architecture is 64- or 32-bit, respectively); if the group is full, further groups are checked using a quadratic probing sequence.

The associated metadata is organized as follows (least significant byte depicted rightmost):

h_i holds information about the i -th bucket of the group:

- 0 if the bucket is empty,
- 1 to signal a *sentinel* (a special value at the end of the bucket array used to finish container iteration).

⁸<https://en.wikipedia.org/wiki/SSE2>

⁹[https://en.wikipedia.org/wiki/ARM_architecture_family#Advanced SIMD_\(Neon\)](https://en.wikipedia.org/wiki/ARM_architecture_family#Advanced SIMD_(Neon))

¹⁰https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm_cmpeq_epi8

¹¹<https://abseil.io/about/design/swisstable>

¹²<https://engineering.fb.com/2019/04/25/developer-tools/f14/>

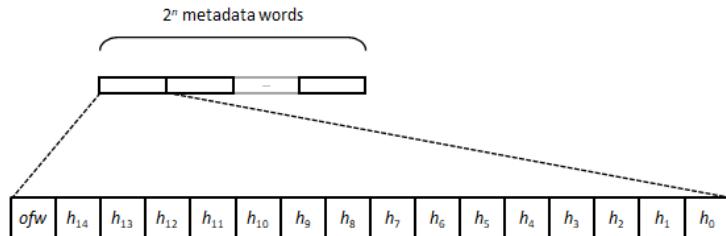


图 59.5: metadata

- otherwise, a reduced hash value in the range [2, 255] obtained from the least significant byte of the element's hash value.

When looking up within a group for an element with hash value h , SIMD operations, if available, are used to match the reduced value of h against the pack of values $\{h_0, h_1, \dots, h_{14}\}$. Locating an empty bucket for insertion is equivalent to matching for 0.

ofw is the so-called *overflow byte*: when inserting an element with hash value h , if the group is full then the $(h \bmod 8)$ -th bit of ofw is set to 1 before moving to the next group in the probing sequence. Lookup probing can then terminate when the corresponding overflow bit is 0. Note that this procedure removes the need to use tombstones.

If neither SSE2 nor Neon is available on the target architecture, the logical organization of metadata stays the same, but information is mapped to two physical 64-bit words using *bit interleaving* as shown in the figure:

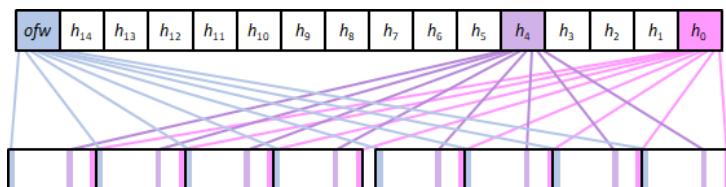


图 59.6: metadata_interleaving

Bit interleaving allows for a reasonably fast implementation of matching operations in the absence of SIMD.

59.3 Rehashing

The maximum load factor of `boost::unordered_flat_map` is 0.875 and can't be changed by the user. As discussed previously, non-relocating open addressing has the problem that average probe length doesn't decrease on deletion when the erased elements are in mid-sequence: so, continuously inserting and erasing elements without triggering a rehash will slowly degrade the container's performance; we call this phenomenon *drifting*. `boost::unordered_flat_map` introduces the following anti-drift mechanism: rehashing is controlled by the container's *maximum load*, initially 0.875 times the size of the bucket array; when erasing an element whose associated overflow bit is not zero, the maximum load is decreased by one. Anti-drift guarantees that rehashing will be eventually triggered in a scenario of repeated insertions and deletions.

59.4 Hash post-mixing

It is well known that open-addressing containers require that the hash function be of good quality, in the sense that close input values (for some natural notion of closeness) are mapped to distant hash values. In particular, a hash function is said to have the *avalanching property*¹³ if flipping a bit in the physical representation of the input changes all bits of the output value with probability 50%. Note that avalanching hash functions are extremely well behaved, and less stringent behaviors are generally good enough in most open-addressing scenarios.

Being a general-purpose container, `boost::unordered_flat_map` does not impose any condition on the user-provided hash function beyond what is required by the C++ standard for unordered associative containers. In order to cope with poor-quality hash functions (such as the identity for integral types), an automatic bit-mixing stage is added to hash values:

- 64-bit architectures: we use the `xmx` function defined in Jon Maiga's "The construct of a bit mixer"¹⁴.
- 32-bit architectures: the chosen mixer has been automatically generated by Hash Function Prospector¹⁵ and selected as the best overall performer in internal benchmarks. Score assigned by Hash Prospector: 333.7934929677524.

There's an opt-out mechanism available to end users so that avalanching hash functions can be marked as such and thus be used without post-mixing. In particular, the specializations of `boost::hash` for string types are marked as avalanching.

59.5 Statistical properties of `boost::unordered_flat_map`

We have written a simulation program¹⁶ to calculate some statistical properties of `boost::unordered_flat_map` as compared with Abseil's `absl::flat_hash_map`, which is generally regarded as one of the fastest hash containers available. For the purposes of this analysis, the main design characteristics of `absl::flat_hash_map` are:

- Bucket array sizes are of the form 2^n , $n \geq 4$.
- Hash mapping is done at the bucket level (rather than at the group level as in `boost::unordered_flat_map`).
- Metadata consists of one byte per bucket, where the most significant bit is set to 1 if the bucket is empty, deleted (tombstone) or a sentinel. The remaining 7 bits hold the reduced hash value for occupied buckets.
- Lookup/insertion uses SIMD to inspect the 16 contiguous buckets beginning at the hash-mapped position, and then continues with further 16-bucket groups using quadratic probing. Probing ends when a non-full group is found. Note that the start positions of these groups are not aligned modulo 16.

The figure shows:

¹³https://en.wikipedia.org/wiki/Avalanche_effect

¹⁴<http://jonkagstrom.com/bit-mixer-construction/index.html>

¹⁵<https://github.com/skeeto/hash-prospector>

¹⁶https://github.com/joaquintides/boost_unordered_flat_map_stats

- the probability that a randomly selected group is full,
- the average number of hops (i.e. the average probe length minus one) for successful and unsuccessful lookup

as functions of the load factor, with perfectly random input and without intervening deletions. Solid line is `boost::unordered_flat_map`, dashed line is `absl::flat_hash_map`.

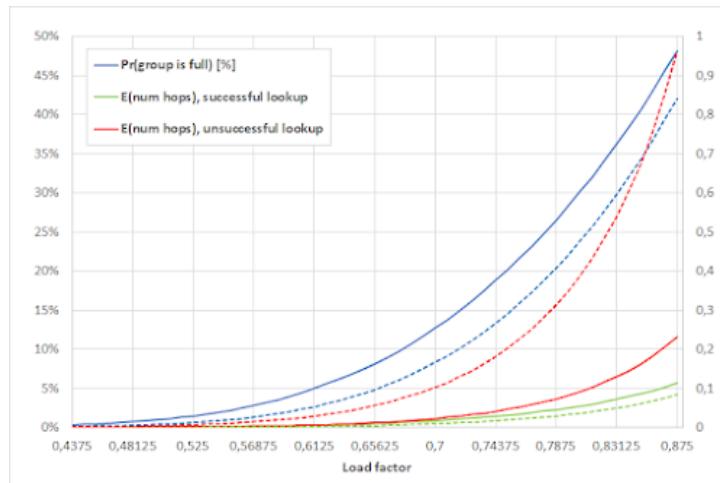


图 59.7: stats1

Some observations:

- $Pr(\text{group is full})$ is higher for `boost::unordered_flat_map`. This follows from the fact that free buckets cluster at the end of 15-aligned groups, whereas for `absl::flat_hash_map` free buckets are uniformly distributed across the array, which increases the probability that a contiguous 16-bucket chunk contains at least one free position. Consequently, $E(\text{num hops})$ for successful lookup is also higher in `boost::unordered_flat_map`.
- By contrast, $E(\text{num hops})$ for unsuccessful lookup is considerably lower in `boost::unordered_flat_map`: `absl::flat_hash_map` uses an all-or-nothing condition for probe termination (group is non-full/full), whereas `boost::unordered_flat_map` uses the 8 bits of information in the overflow byte to allow for more finely-grained termination —effectively, making probe termination 1.75 times more likely. The overflow byte acts as a sort of Bloom filter¹⁷ to check for probe termination based on reduced hash value.

The next figure shows the average number of actual comparisons (i.e. when the reduced hash value matched) for successful and unsuccessful lookup. Again, solid line is `boost::unordered_flat_map` and dashed line is `absl::flat_hash_map`.

$E(\text{num cmps})$ is a function of:

- $E(\text{num hops})$ (lower better),
- the size of the group (lower better),
- the number of bits of the reduced hash value (higher better).

¹⁷https://en.wikipedia.org/wiki/Bloom_filter

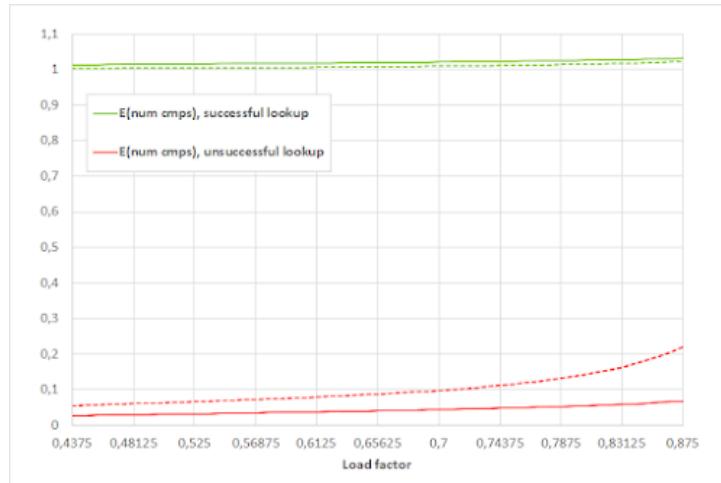


图 59.8: stats2

We see then that `boost::unordered_flat_map` approaches `absl::flat_hash_map` on $E(\text{num cmps})$ for successful lookup (1% higher or less), despite its poorer $E(\text{num hops})$ figures: this is so because `boost::unordered_flat_map` uses smaller groups (15 vs. 16) and, most importantly, because its reduced hash values contain $\log_2(254) = 7.99$ bits vs. 7 bits in `absl::flat_hash_map`, and each additional bit in the hash reduced value decreases the number of negative comparisons roughly by half. In the case of $E(\text{num cmps})$ for unsuccessful lookup, `boost::unordered_flat_map` figures are up to 3.2 times lower under high-load conditions.

59.6 Benchmarks

59.6.1 Running-n plots

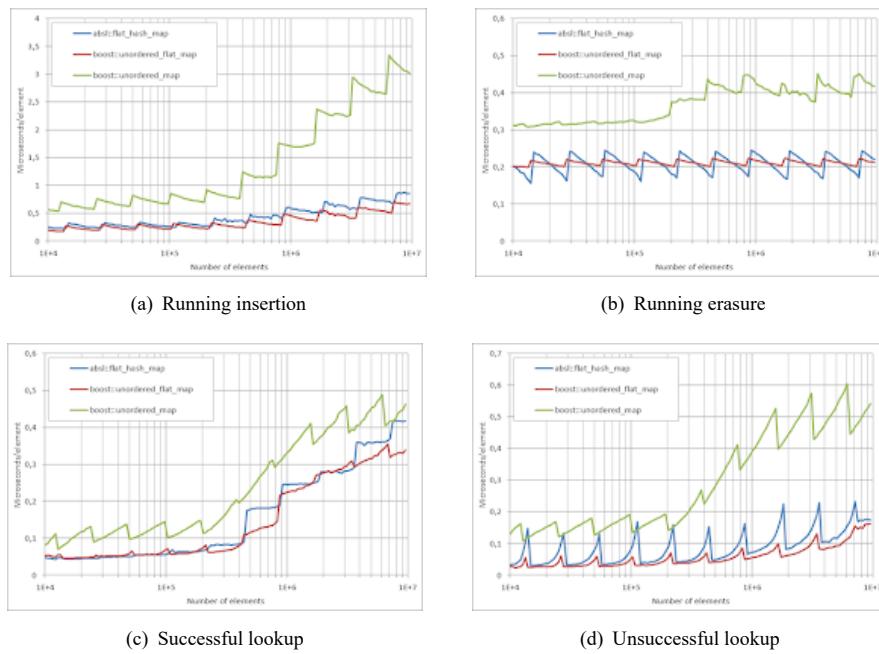
We have measured the execution times of `boost::unordered_flat_map` against `absl::flat_hash_map` and `boost::unordered_map` for basic operations (insertion, erasure during iteration, successful lookup, unsuccessful lookup) with container size n ranging from 10,000 to 10M. We provide the full benchmark code and results for different 64- and 32-bit architectures in a dedicated repository¹⁸; here, we just show the plots for GCC 11 in x64 mode on an AMD EPYC Rome 7302P @ 3.0GHz. Please note that each container uses its own default hash function, so a direct comparison of execution times may be slightly biased.

As predicted by our statistical analysis, `boost::unordered_flat_map` is considerably faster than `absl::flat_hash_map` for unsuccessful lookup because the average probe length and number of (negative) comparisons are much lower; this effect translates also to insertion, since `insert` needs to first check that the element is not present, so it internally performs an unsuccessful lookup. Note how performance is less impacted (stays flatter) when the load factor increases.

As for successful lookup, `boost::unordered_flat_map` is still faster, which may be due to its better cache locality, particularly for low load factors: in this situation, elements are clustered at the beginning portion of each group, while for `absl::flat_hash_map` they are uniformly distributed with more empty space in between.

`boost::unordered_flat_map` is slower than `absl::flat_hash_map` for running erasure (erasure of some elements during container traversal). The actual culprit here is iteration, which is particularly slow; this is a collateral effect of having SIMD operations work only on 16-aligned metadata words, while `absl::flat_hash_map`

¹⁸https://github.com/boostorg/boost_unordered_benchmarks/tree/boost_unordered_flat_map



iteration looks ahead 16 metadata bytes beyond the current iterator position.

59.6.2 Aggregate performance

Boost.Unordered provides a series of benchmarks¹⁹ emulating real-life scenarios combining several operations for a number of hash containers and key types (`std::string`, `std::string_view`, `std::uint32_t`, `std::uint64_t` and a UUID class of size 16). The interested reader can build and run the benchmarks on her environment of choice; as an example, these are the results for GCC 11 in x64 mode on an Intel Xeon E5-2683 @ 2.10GHz:

`std::string`

```

    std::unordered_map: 38021 ms, 175723032 bytes in 3999509 allocations
    boost::unordered_map: 30785 ms, 149465712 bytes in 3999510 allocations
    boost::unordered_flat_map: 14486 ms, 134217728 bytes in 1 allocations
        multi_index_map: 30162 ms, 178316048 bytes in 3999510 allocations
        absl::node_hash_map: 15403 ms, 139489608 bytes in 3999509 allocations
        absl::flat_hash_map: 13018 ms, 142606336 bytes in 1 allocations
    std::unordered_map, FNV-1a: 43893 ms, 175723032 bytes in 3999509 allocations
    boost::unordered_map, FNV-1a: 33730 ms, 149465712 bytes in 3999510 allocations
boost::unordered_flat_map, FNV-1a: 15541 ms, 134217728 bytes in 1 allocations
        multi_index_map, FNV-1a: 33915 ms, 178316048 bytes in 3999510 allocations
        absl::node_hash_map, FNV-1a: 20701 ms, 139489608 bytes in 3999509 allocations
        absl::flat_hash_map, FNV-1a: 18234 ms, 142606336 bytes in 1 allocations

```

`std::string_view`

```

    std::unordered_map: 38481 ms, 207719096 bytes in 3999509 allocations
    boost::unordered_map: 26066 ms, 181461776 bytes in 3999510 allocations

```

¹⁹<https://github.com/boostorg/unordered/tree/develop/benchmark>

```

boost::unordered_flat_map: 14923 ms, 197132280 bytes in 1 allocations
    multi_index_map: 27582 ms, 210312120 bytes in 3999510 allocations
    absl::node_hash_map: 14670 ms, 171485672 bytes in 3999509 allocations
    absl::flat_hash_map: 12966 ms, 209715192 bytes in 1 allocations
    std::unordered_map, FNV-1a: 45070 ms, 207719096 bytes in 3999509 allocations
    boost::unordered_map, FNV-1a: 29148 ms, 181461776 bytes in 3999510 allocations
boost::unordered_flat_map, FNV-1a: 15397 ms, 197132280 bytes in 1 allocations
    multi_index_map, FNV-1a: 30371 ms, 210312120 bytes in 3999510 allocations
    absl::node_hash_map, FNV-1a: 19251 ms, 171485672 bytes in 3999509 allocations
    absl::flat_hash_map, FNV-1a: 17622 ms, 209715192 bytes in 1 allocations

```

std::uint32_t

```

    std::unordered_map: 21297 ms, 192888392 bytes in 5996681 allocations
    boost::unordered_map: 9423 ms, 149424400 bytes in 5996682 allocations
boost::unordered_flat_map: 4974 ms, 71303176 bytes in 1 allocations
    multi_index_map: 10543 ms, 194252104 bytes in 5996682 allocations
    absl::node_hash_map: 10653 ms, 123470920 bytes in 5996681 allocations
    absl::flat_hash_map: 6400 ms, 75497480 bytes in 1 allocations

```

std::uint64_t

```

    std::unordered_map: 21463 ms, 240941512 bytes in 6000001 allocations
    boost::unordered_map: 10320 ms, 197477520 bytes in 6000002 allocations
boost::unordered_flat_map: 5447 ms, 134217728 bytes in 1 allocations
    multi_index_map: 13267 ms, 242331792 bytes in 6000002 allocations
    absl::node_hash_map: 10260 ms, 171497480 bytes in 6000001 allocations
    absl::flat_hash_map: 6530 ms, 142606336 bytes in 1 allocations

```

uuid

```

    std::unordered_map: 37338 ms, 288941512 bytes in 6000001 allocations
    boost::unordered_map: 24638 ms, 245477520 bytes in 6000002 allocations
boost::unordered_flat_map: 9223 ms, 197132280 bytes in 1 allocations
    multi_index_map: 25062 ms, 290331800 bytes in 6000002 allocations
    absl::node_hash_map: 14005 ms, 219497480 bytes in 6000001 allocations
    absl::flat_hash_map: 10559 ms, 209715192 bytes in 1 allocations

```

Each container uses its own default hash function, except the entries labeled FNV-1a in `std::string` and `std::string_view`, which use the same implementation of Fowler-Noll-Vo hash, version 1a²⁰, and `uuid`, where all containers use the same user-provided function based on `boost::hash_combine`²¹.

²⁰https://en.wikipedia.org/wiki/Fowler%20%93Noll%20%93Vo_hash_function#FNV-1a_hash

²¹https://www.boost.org/libs/container_hash/doc/html/hash.html#ref_hash_combine

59.7 Deviations from the standard

The adoption of open addressing imposes a number of deviations from the C++ standard for unordered associative containers. Users should keep them in mind when migrating to `boost::unordered_flat_map` from `boost::unordered_map` (or from any other implementation of `std::unordered_map`):

- Both Key and T in `boost::unordered_flat_map<Key,T>` must be MoveConstructible²². This is due to the fact that elements are stored directly into the bucket array and have to be transferred to a new block of memory on rehashing; by contrast, `boost::unordered_map` is a node-based container and elements are never moved once constructed.
- For the same reason, pointers and references to elements become invalid after rehashing (`boost::unordered_map` only invalidates iterators).
- `begin()` is not constant-time (the bucket array is traversed till the first non-empty bucket is found).
- `erase(iterator)` returns `void` rather than an iterator to the element after the erased one. This is done to maximize performance, as locating the next element requires traversing the bucket array; if that element is absolutely required, the `erase(iterator++)` idiom can be used. This performance issue is not exclusive to open addressing, and has been discussed²³ in the context of the C++ standard too.
- The maximum load factor can't be changed by the user (`max_load_factor(z)` is provided for backwards compatibility reasons, but does nothing). Rehashing can occur *before* the load reaches `max_load_factor() * bucket_count()` due to the anti-drift mechanism described previously.
- There is no bucket API (`bucket_size`, `begin(n)`, etc.) save `bucket_count`.
- There are no node handling facilities (`extract`²⁴, etc.) Such functionality makes no sense here as open-addressing containers are precisely *not* node-based. `merge`²⁵ is provided, but the implementation relies on element movement rather than node transferring.

59.8 Conclusions and next steps

`boost::unordered_flat_map` and `boost::unordered_flat_set` are the new open-addressing containers in Boost.Unordered providing top speed in exchange for some interface and behavioral deviations from the standards-compliant `boost::unordered_map` and `boost::unordered_set`. We have analyzed their internal data structure and provided some theoretical and practical evidence for their excellent performance. As of this writing, we claim `boost::unordered_flat_map/boost::unordered_flat_set` to rank among the fastest hash containers available to C++ programmers.

With this work, we have reached an important milestone in the ongoing Development Plan for Boost.Unordered²⁶. After Boost 1.81, we will continue improving the functionality and performance of existing containers and will possibly augment the available container catalog to offer greater freedom of choice to Boost users. Your feedback on our current and future work is much welcome.

²²https://en.cppreference.com/w/cpp/named_req/MoveConstructible

²³<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2023.pdf>

²⁴https://en.cppreference.com/w/cpp/container/unordered_map/extract

²⁵https://en.cppreference.com/w/cpp/container/unordered_map/merge

²⁶https://pdimov.github.io/articles/unordered_dev_plan.html

Chapter 60

Static B-Trees

Algorithmica   ★★

This section is a follow-up to the previous one¹, where we optimized binary search by the means of removing branching and improving the memory layout. Here, we will also be searching in sorted arrays, but this time we are not limited to fetching and comparing only one element at a time.

In this section, we generalize the techniques we developed for binary search to static B-trees and accelerate them further using SIMD instructions². In particular, we develop two new implicit data structures:

- The first³ is based on the memory layout of a B-tree, and, depending on the array size, it is up to 8x faster than `std::lower_bound` while using the same space as the array and only requiring a permutation of its elements.
- The second⁴ is based on the memory layout of a B+ tree, and it is up to 15x faster than `std::lower_bound` while using just 6-7% more memory —or 6-7% of the memory if we can keep the original sorted array.

To distinguish them from B-trees —the structures with pointers, hundreds to thousands of keys per node, and empty spaces in them —we will use the names S-tree and S+ tree respectively to refer to these particular memory layouts.

Similar to B-trees⁵, “the more you think about what the S in S-trees means, the better you understand S-trees.”

To the best of my knowledge, this is a significant improvement over the existing approaches⁶. As before, we are using Clang 10 targeting a Zen 2 CPU, but the performance improvements should approximately transfer to most other platforms, including Arm-based chips. Use this single-source benchmark⁷ of the final implementation if you want to test it on your machine.

¹<https://en.algorithmica.org/hpc/data-structures/binary-search>

²<https://en.algorithmica.org/hpc/simd>

³<https://en.algorithmica.org/hpc/data-structures/s-tree/#b-tree-layout>

⁴<https://en.algorithmica.org/hpc/data-structures/s-tree/#b-tree-layout-1>

⁵<https://en.wikipedia.org/wiki/B-tree#Origin>

⁶http://kaldevey.com/pubs/FAST_SIGMOD10.pdf

⁷<https://github.com/sslotin/amh-code/blob/main/binsearch/standalone.cc>

This is a long article, and since it also serves as a textbook⁸ case study, we will improve the algorithm incrementally for pedagogical goals. If you are already an expert and feel comfortable reading intrinsic⁹-heavy code with little to no context, you can jump straight to the final implementation¹⁰.

60.1 B-Tree Layout

B-trees generalize the concept of binary search trees by allowing nodes to have more than two children. Instead of a single key, a node of a B-tree of order k can contain up to $B = (k-1)$ keys stored in sorted order and up to k pointers to child nodes. Each child i satisfies the property that all keys in its subtree are between keys $(i-1)$ and i of the parent node (if they exist).

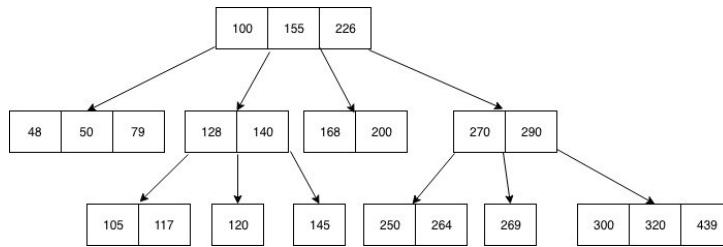


图 60.1: A B-tree of order 4

The main advantage of this approach is that it reduces the tree height by $\frac{\log 2n}{\log kn} = \frac{\log_2 k}{\log_2 2} = \log 2k$ times, while fetching each node still takes roughly the same time —as long it fits into a single memory block¹¹.

B-trees were primarily developed for the purpose of managing on-disk databases, where the latency of randomly fetching a single byte is comparable with the time it takes to read the next 1MB of data sequentially. For our use case, we will be using the block size of $B = 16$ elements —or 64 bytes, the size of the cache line —which makes the tree height and the total number of cache line fetches per query $\log 217 \approx 4$ times smaller compared to the binary search.

60.2 Implicit B-Tree

Storing and fetching pointers in a B-tree node wastes precious cache space and decreases performance, but they are essential for changing the tree structure on inserts and deletions. But when there are no updates and the structure of a tree is static, we can get rid of the pointers, which makes the structure *implicit*.

One of the ways to achieve this is by generalizing the Eytzinger numeration¹² to $(B + 1)$ -ary trees:

- The root node is numbered 0.
- Node k has $(B + 1)$ child nodes numbered $k(B + 1) + i + 1$ for $i \in [0, B]$.

This way, we can only use $O(1)$ additional memory by allocating one large two-dimensional array of keys and relying on index arithmetic to locate children nodes in the tree:

⁸<https://en.algorithmica.org/hpc/>

⁹<https://en.algorithmica.org/hpc/simd/intrinsics>

¹⁰<https://en.algorithmica.org/hpc/data-structures/s-tree/#implicit-b-tree-1>

¹¹<https://en.algorithmica.org/hpc/external-memory/hierarchy/>

¹²<https://en.algorithmica.org/hpc/data-structures/binary-search#eytzinger-layout>

```

const int B = 16;

int nblocks = (n + B - 1) / B;
int btree[nblocks][B];

int go(int k, int i) { return k * (B + 1) + i + 1; }

```

This numeration automatically makes the B-tree complete or almost complete with the height of $\Theta(\log_{B+1} n)$. If the length of the initial array is not a multiple of B , the last block is padded with the largest value of its data type.

60.3 Construction

We can construct the B-tree similar to how we constructed the Eytzinger array—by traversing the search tree:

```

void build(int k = 0) {
    static int t = 0;
    if (k < nblocks) {
        for (int i = 0; i < B; i++) {
            build(go(k, i));
            btree[k][i] = (t < n ? a[t++] : INT_MAX);
        }
        build(go(k, B));
    }
}

```

It is correct because each value of the initial array will be copied to a unique position in the resulting array, and the tree height is $\Theta(\log_{B+1} n)$ because k is multiplied by $(B + 1)$ each time we descend into a child node.

Note that this numeration causes a slight imbalance: left-er children may have larger subtrees, although this is only true for $O(\log_{B+1} n)$ parent nodes.

60.4 Searches

To find the lower bound, we need to fetch the B keys in a node, find the first key a_i not less than x , descend to the i -th child—and continue until we reach a leaf node. There is some variability in how to find that first key. For example, we could do a tiny internal binary search that makes $O(\log B)$ iterations, or maybe just compare each key sequentially in $O(B)$ time until we find the local lower bound, hopefully exiting from the loop a bit early.

But we are not going to do that—because we can use SIMD¹³. It doesn’t work well with branching, so essentially what we want to do is to compare against all B elements regardless, compute a bitmask out of these comparisons, and then use the **ffs** instruction to find the bit corresponding to the first non-lesser element:

¹³<https://en.algorithmica.org/hpc/simd>

```

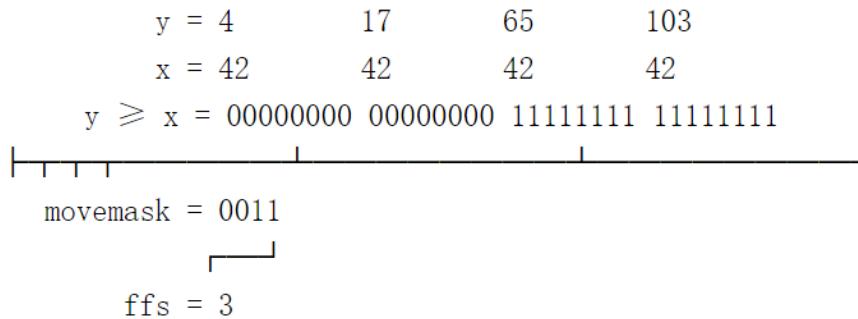
int mask = (1 << B);

for (int i = 0; i < B; i++)
    mask |= (btree[k][i] >= x) << i;

int i = __builtin_ffs(mask) - 1;
// now i is the number of the correct child node

```

Unfortunately, the compilers are not smart enough to auto-vectorize¹⁴ this code yet, so we have to optimize it manually. In AVX2, we can load 8 elements, compare them against the search key, producing a vector mask¹⁵, and then extract the scalar mask from it with `movemask`. Here is a minimized illustrated example of what we want to do:



Since we are limited to processing 8 elements at a time (half our block / cache line size), we have to split the elements into two groups and then combine the two 8-bit masks. To do this, it will be slightly easier to swap the condition for $x > y$ and compute the inverted mask instead:

```

typedef __m256i reg;

int cmp(reg x_vec, int* y_ptr) {
    reg y_vec = _mm256_load_si256((reg*) y_ptr); // load 8 sorted elements
    reg mask = _mm256_cmpgt_epi32(x_vec, y_vec); // compare against the key
    return _mm256_movemask_ps((__m256) mask); // extract the 8-bit mask
}

```

Now, to process the entire block, we need to call it twice and combine the masks:

```

int mask = ~(
    cmp(x, &btree[k][0]) +
    (cmp(x, &btree[k][8]) << 8)
);

```

To descend down the tree, we use `ffs` on that mask to get the correct child number and just call the `go` function we defined earlier:

¹⁴<https://en.algorithmica.org/hpc/simd/auto-vectorization/>

¹⁵<https://en.algorithmica.org/hpc/simd/masking/>

```
int i = __builtin_ffs(mask) - 1;
k = go(k, i);
```

To actually return the result in the end, we'd want to just fetch `btree[k][i]` in the last node we visited, but the problem is that sometimes the local lower bound doesn't exist ($i \geq B$) because x happens to be greater than all the keys in the node. We could, in theory, do the same thing we did for the Eytzinger binary search¹⁶ and restore the correct element *after* we calculate the last index, but we don't have a nice bit trick this time and have to do a lot of divisions by 17¹⁷ to compute it, which will be slow and almost certainly not worth it.

Instead, we can just remember and return the last local lower bound we encountered when we descended the tree:

```
int lower_bound(int _x) {
    int k = 0, res = INT_MAX;
    reg x = _mm256_set1_epi32(_x);
    while (k < nbblocks) {
        int mask = ~(
            cmp(x, &btree[k][0]) +
            (cmp(x, &btree[k][8]) << 8)
        );
        int i = __builtin_ffs(mask) - 1;
        if (i < B)
            res = btree[k][i];
        k = go(k, i);
    }
    return res;
}
```

This implementation outperforms all previous binary search implementations, and by a huge margin: This is very good—but we can optimize it even further.

60.5 Optimization

Before everything else, let's allocate the memory for the array on a hugepage¹⁸:

```
const int P = 1 << 21; // page size in bytes (2MB)
const int T = (64 * nbblocks + P - 1) / P * P; // can only allocate whole number of pages
btree = (int(*)[16]) std::aligned_alloc(P, T);
madvise(btree, T, MADV_HUGEPAGE);
```

This slightly improves the performance on larger array sizes:

Ideally, we'd also need to enable hugepages for all previous implementations¹⁹ to make the comparison

¹⁶<https://en.algorithmica.org/hpc/data-structures/binary-search/#search-implementation>

¹⁷<https://en.algorithmica.org/hpc/arithmetic/division>

¹⁸<https://en.algorithmica.org/hpc/cpu-cache/paging>

¹⁹<https://en.algorithmica.org/hpc/data-structures/binary-search>

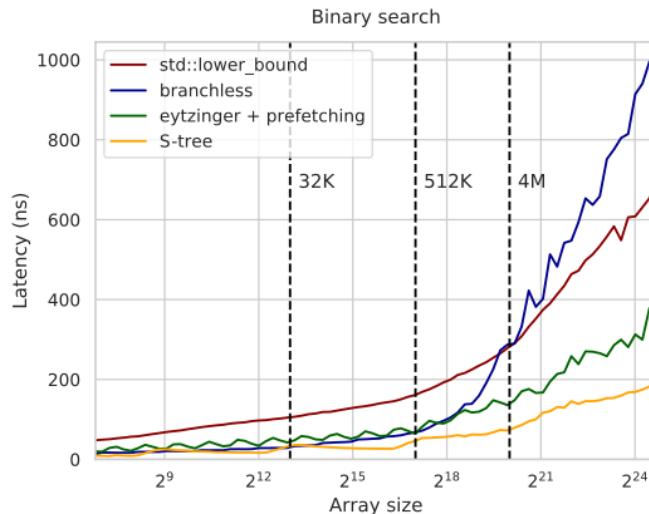


图 60.2: Binary search

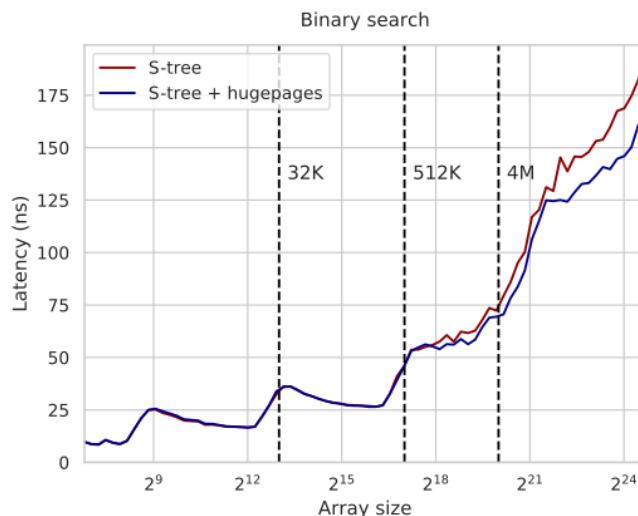


图 60.3: Binary search

fair, but it doesn't matter that much because they all have some form of prefetching that alleviates this problem.

With that settled, let's begin real optimization. First of all, we'd want to use compile-time constants instead of variables as much as possible because it lets the compiler embed them in the machine code, unroll loops, optimize arithmetic, and do all sorts of other nice stuff for us for free. Specifically, we want to know the tree height in advance:

```
constexpr int height(int n) {
    // grow the tree until its size exceeds n elements
    int s = 0, // total size so far
    l = B, // size of the next layer
    h = 0; // height so far
    while (s + l - B < n) {
```

```

    s += 1;
    l *= (B + 1);
    h++;
}
return h;
}

const int H = height(N);

```

Next, we can find the local lower bound in nodes faster. Instead of calculating it separately for two 8-element blocks and merging two 8-bit masks, we combine the vector masks using the `packs`²⁰ instruction and readily extract it using `movemask` just once:

```

unsigned rank(reg x, int* y) {
    reg a = _mm256_load_si256((reg*) y);
    reg b = _mm256_load_si256((reg*) (y + 8));

    reg ca = _mm256_cmpgt_epi32(a, x);
    reg cb = _mm256_cmpgt_epi32(b, x);

    reg c = _mm256_packs_epi32(ca, cb);
    int mask = _mm256_movemask_epi8(c);

    // we need to divide the result by two because we call movemask_epi8 on 16-bit masks:
    return __tzcnt_u32(mask) >> 1;
}

```

This instruction converts 32-bit integers stored in two registers to 16-bit integers stored in one register — in our case, effectively joining the vector masks into one. Note that we've swapped the order of comparison — this lets us not invert the mask in the end, but we have to subtract one from the search key once in the beginning to make it correct (otherwise, it works as `upper_bound`).

If you need to work with floating-point²¹ keys, consider whether `upper_bound` will suffice — because if you need `lower_bound` specifically, then subtracting one or the machine epsilon from the search key doesn't work: you need to get the previous representable number²² instead. Aside from some corner cases, this essentially means reinterpreting its bits as an integer, subtracting one, and reinterpreting it back as a float (which magically works because of how IEEE-754 floating-point numbers²³ are stored in memory).

²⁰https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=3037,4870,6715,4845,3853,90,7307,5993,2692,6946,6949,5456,6938,5456,1021,3007,514,518,7253,7183,3892,5135,5260,3915,4027,3873,7401,4376,4229,151,2324,2310,2324,4075,6130,4875,6385,5259,6385,6250,1395,7253,6452,7492,4669,4669,7253,1039,1029,4669,4707,7253,7242,848,879,848,7251,4275,879,874,849,833,6046,7250,4870,4872,4875,849,849,5144,4875,4787,4787,4787,5227,7359,7335,7392,4787,5259,5230,5223,6438,488,483,6165,6570,6554,289,6792,6554,5230,6385,5260,5259,289,288,3037,3009,590,604,5230,5259,6554,6554,5259,6547,6554,3841,5214,5229,5260,5259,7335,5259,519,1029,515,3009,3009,3011,515,6527,6527,6554,288,3841,5230,5259,5230,5259,305,5259,591,633,633,5259,5230,5259,5259,3017,3018,3037,3017,3017,3016,3013,5144&text=_mm256_packs_epi32&techs=AVX,AVX2

²¹<https://en.algorithmica.org/hpc/arithmetic/float>

²²<https://stackoverflow.com/questions/10160079/how-to-find-nearest-next-previous-double-value-numeric-limitsepsilon-for-give>

²³<https://en.algorithmica.org/hpc/arithmetic/ieee-754>

The problem is, it does this weird interleaving where the result is written in the `a1 b1 a2 b2` order instead of `a1 a2 b1 b2` that we want —many AVX2 instructions tend to do that. To correct this, we need to permute²⁴ the resulting vector, but instead of doing it during the query time, we can just permute every node during preprocessing:

```
void permute(int *node) {
    const reg perm = _mm256_setr_epi32(4, 5, 6, 7, 0, 1, 2, 3);
    reg* middle = (reg*) (node + 4);
    reg x = _mm256_loadu_si256(middle);
    x = _mm256_permutevar8x32_epi32(x, perm);
    _mm256_storeu_si256(middle, x);
}
```

Now we just call `permute(&btree[k])` right after we are done building the node. There are probably faster ways to swap the middle elements, but we will leave it here as the preprocessing time is not that important for now.

This new SIMD routine is significantly faster because the extra `movmask` is slow, and also blending the two masks takes quite a few instructions. Unfortunately, we now can't just do the `res = btree[k][i]` update anymore because the elements are permuted. We can solve this problem with some bit-level trickery in terms of `i`, but indexing a small lookup table turns out to be faster and also doesn't require a new branch:

```
const int translate[17] = {
    0, 1, 2, 3,
    8, 9, 10, 11,
    4, 5, 6, 7,
    12, 13, 14, 15,
    0
};

void update(int &res, int* node, unsigned i) {
    int val = node[translate[i]];
    res = (i < B ? val : res);
}
```

This `update` procedure takes some time, but it's not on the critical path between the iterations, so it doesn't affect the actual performance that much.

Stitching it all together (and leaving out some other minor optimizations):

```
int lower_bound(int _x) {
    int k = 0, res = INT_MAX;
    reg x = _mm256_set1_epi32(_x - 1);
    for (int h = 0; h < H - 1; h++) {
        unsigned i = rank(x, &btree[k]);
        update(res, &btree[k], i);
```

²⁴<https://en.algorithmica.org/hpc/simd/shuffling>

```

    k = go(k, i);
}

// the last branch:
if (k < nblocks) {
    unsigned i = rank(x, btree[k]);
    update(res, &btree[k], i);
}
return res;
}

```

All this work saved us 15-20% or so:

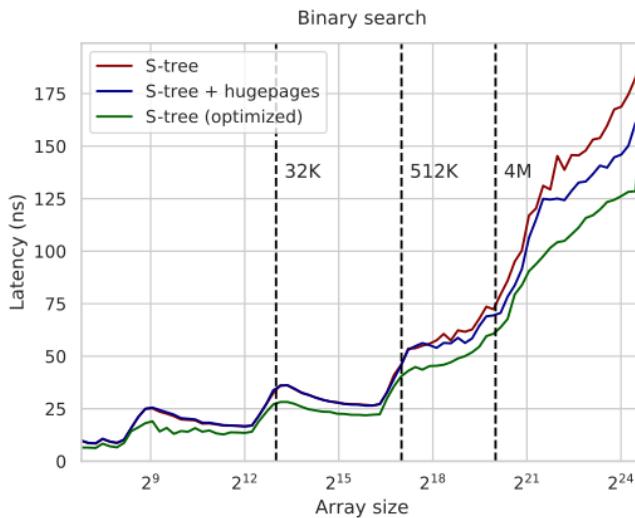


图 60.4: Binary search

It doesn't feel very satisfying so far, but we will reuse these optimization ideas later.

There are two main problems with the current implementation:

- The `update` procedure is quite costly, especially considering that it is very likely going to be useless: 16 out of 17 times, we can just fetch the result from the last block.
- We do a non-constant number of iterations, causing branch prediction problems similar to how it did for the Eytzinger binary search²⁵; you can also see it on the graph this time, but the latency bumps have a period of 2^4 .

To address these problems, we need to change the layout a little bit.

60.6 B+ Tree Layout

Most of the time, when people talk about B-trees, they really mean B+ trees, which is a modification that distinguishes between the two types of nodes:

- *Internal nodes* store up to B keys and $(B + 1)$ pointers to child nodes. The key number i is always equal to the smallest key in the subtree of the $(i + 1)$ -th child node.

²⁵<https://en.algorithmica.org/hpc/data-structures/binary-search/#removing-the-last-branch>

- *Data nodes* or *leaves* store up to BB keys, the pointer to the next leaf node, and, optionally, an associated value for each key —if the structure is used as a key-value map.

The advantages of this approach include faster search time (as the internal nodes only store keys) and the ability to quickly iterate over a range of entries (by following next leaf node pointers), but this comes at the cost of some memory overhead: we have to store copies of keys in the internal nodes.

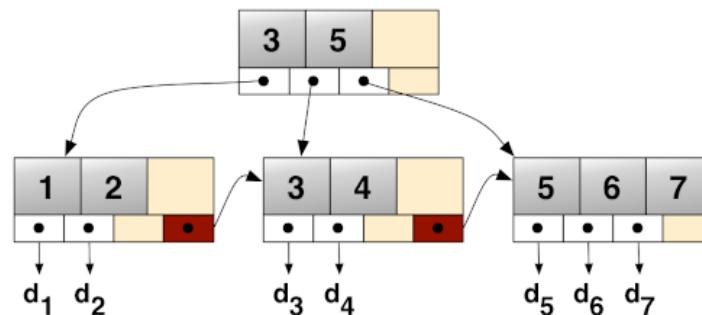


图 60.5: A B+ tree of order 4

Back to our use case, this layout can help us solve our two problems:

- Either the last node we descend into has the local lower bound, or it is the first key of the next leaf node, so we don't need to call `update` on each iteration.
- The depth of all leaves is constant because B+ trees grow at the root and not at the leaves, which removes the need for branching.

The disadvantage is that this layout is not succinct: we need some additional memory to store the internal nodes —about $\frac{1}{16}$ -th of the original array size, to be exact—but the performance improvement will be more than worth it.

60.7 Implicit B+ Tree

To be more explicit with pointer arithmetic, we will store the entire tree in a single one-dimensional array. To minimize index computations during run time, we will store each layer sequentially in this array and use compile time computed offsets to address them: the keys of the node number `k` on layer `h` start with `btree[offset(h) + k * B]`, and its `i`-th child will at `btree[offset(h - 1) + (k * (B + 1) + i) * B]`.

To implement all that, we need slightly more `constexpr` functions:

```
// number of B-element blocks in a layer with n keys
constexpr int blocks(int n) {
    return (n + B - 1) / B;
}

// number of keys on the layer previous to one with n keys
constexpr int prev_keys(int n) {
    return (blocks(n) + B) / (B + 1) * B;
}
```

```

// height of a balanced n-key B+ tree
constexpr int height(int n) {
    return (n <= B ? 1 : height(prev_keys(n)) + 1);
}

// where the layer h starts (layer 0 is the largest)
constexpr int offset(int h) {
    int k = 0, n = N;
    while (h--) {
        k += blocks(n) * B;
        n = prev_keys(n);
    }
    return k;
}

const int H = height(N);
const int S = offset(H); // the tree size is the offset of the (non-existent) layer H

int *btree; // the tree itself is stored in a single hugepage-aligned array of size S

```

Note that we store the layers in reverse order, but the nodes within a layer and data in them are still left-to-right, and also the layers are numbered bottom-up: the leaves form the zeroth layer, and the root is the layer $H - 1$. These are just arbitrary decisions — it is just slightly easier to implement in code.

60.8 Construction

To construct the tree from a sorted array a , we first need to copy it into the zeroth layer and pad it with infinities:

```

memcpy(btree, a, 4 * N);

for (int i = N; i < S; i++)
    btree[i] = INT_MAX;

```

Now we build the internal nodes, layer by layer. For each key, we need to descend to the right of it in, always go left until we reach a leaf node, and then take its first key —it will be the smallest on the subtree:

```

for (int h = 1; h < H; h++) {
    for (int i = 0; i < offset(h + 1) - offset(h); i++) {
        // i = k * B + j
        int k = i / B,
            j = i - k * B;
        k = k * (B + 1) + j + 1; // compare to the right of the key
        // and then always to the left
    }
}

```

```

    for (int l = 0; l < h - 1; l++)
        k *= (B + 1);
    // pad the rest with infinities if the key doesn't exist
    btree[offset(h) + i] = (k * B < N ? btree[k * B] : INT_MAX);
}
}

```

And just the finishing touch —we need to permute keys in internal nodes to search them faster:

```

for (int i = offset(1); i < S; i += B)
    permute(btree + i);

```

We start from `offset(1)`, and we specifically don’t permute leaf nodes and leave the array in the original sorted order. The motivation is that we’d need to do this complex index translation we do in `update` if the keys were permuted, and it is on the critical path when this is the last operation. So, just for this layer, we switch to the original mask-blending local lower bound procedure.

60.9 Searching

The search procedure becomes simpler than for the B-tree layout: we don’t need to do `update` and only execute a fixed number of iterations —although the last one with some special treatment:

```

int lower_bound(int _x) {
    unsigned k = 0; // we assume k already multiplied by B to optimize pointer arithmetic
    reg x = _mm256_set1_epi32(_x - 1);
    for (int h = H - 1; h > 0; h--) {
        unsigned i = permuted_rank(x, btree + offset(h) + k);
        k = k * (B + 1) + i * B;
    }
    unsigned i = direct_rank(x, btree + k);
    return btree[k + i];
}

```

Switching to the B+ layout more than paid off: the S+ tree is 1.5-3x faster compared to the optimized S-tree:

The spikes at the high end of the graph are caused by the L1 TLB not being large enough: it has 64 entries, so it can handle at most $64 \times 2 = 128\text{MB}$ of data, which is exactly what is required for storing 2^{25} integers. The S+ tree hits this limit slightly sooner because of the 7% memory overhead.

60.10 Comparison with std::lower_bound

We’ve come a long way from binary search:

On these scales, it makes more sense to look at the relative speedup:

The cliffs at the beginning of the graph are because the running time of `std::lower_bound` grows smoothly with the array size, while for an S+ tree, it is locally flat and increases in discrete steps when a new layer needs to be added.

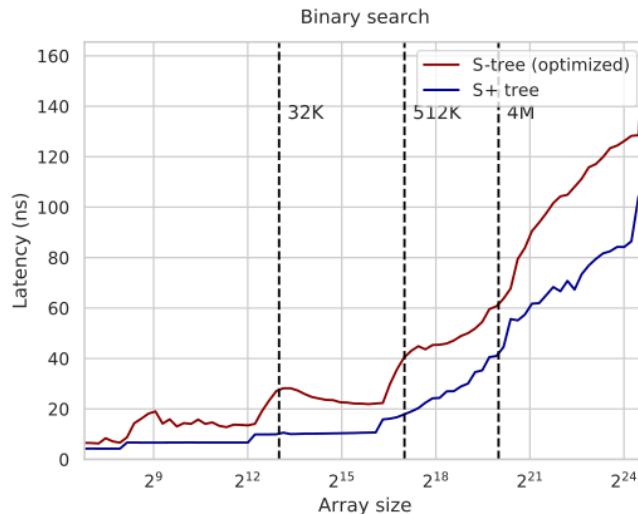


图 60.6: search-bplus

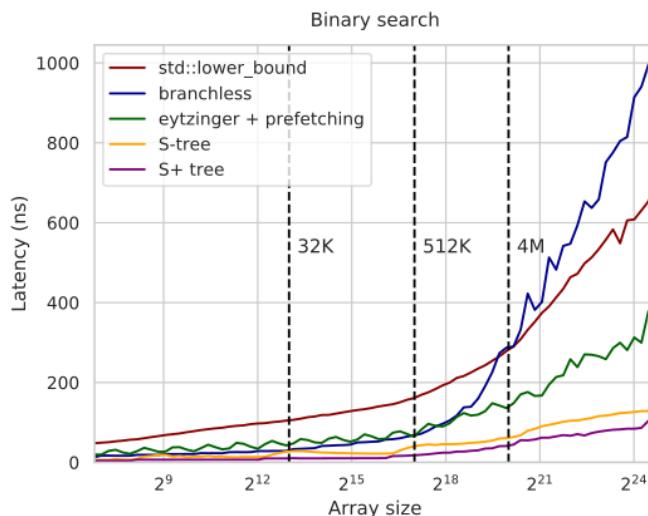


图 60.7: search-all

One important asterisk we haven't discussed is that what we are measuring is not real latency, but the *reciprocal throughput* —the total time it takes to execute a lot of queries divided by the number of queries:

```
clock_t start = clock();

for (int i = 0; i < m; i++)
    checksum ^= lower_bound(q[i]);

float seconds = float(clock() - start) / CLOCKS_PER_SEC;
printf("%.2f ns per query\n", 1e9 * seconds / m);
```

To measure *actual* latency, we need to introduce a dependency between the loop iterations so that the next query can't start before the previous one finishes:

```
int last = 0;
```

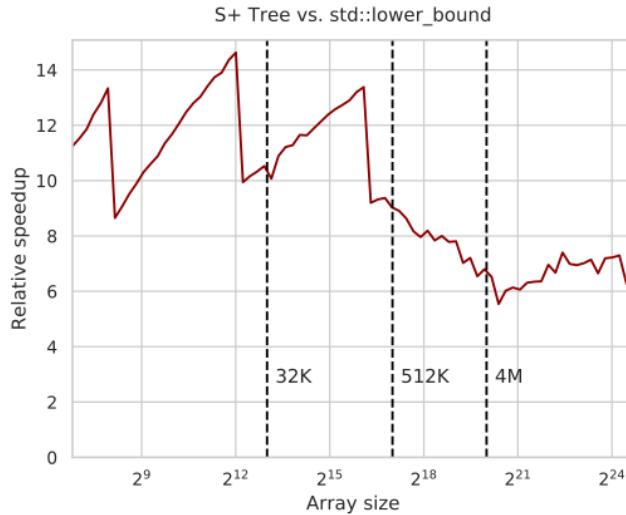


图 60.8: search-relative

```

for (int i = 0; i < m; i++) {
    last = lower_bound(q[i] ^ last);
    checksum ^= last;
}

```

In terms of real latency, the speedup is not that impressive:

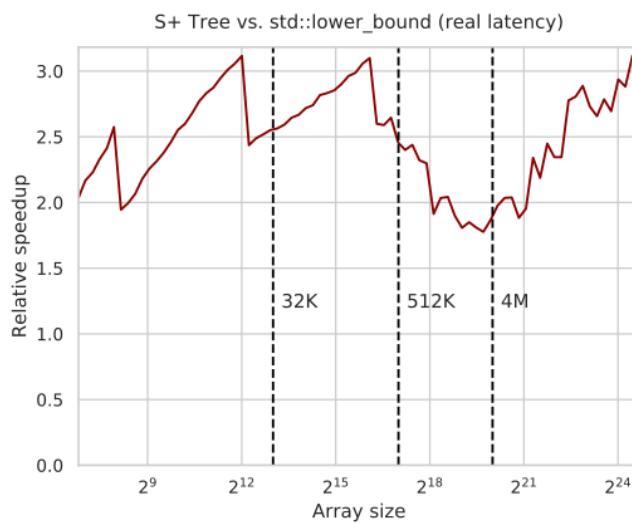


图 60.9: search-relative-latency

A lot of the performance boost of the S+ tree comes from removing branching and minimizing memory requests, which allows overlapping the execution of more adjacent queries —apparently, around three on average.

Although nobody except maybe the HFT people cares about real latency, and everybody actually measures throughput even when using the word “latency,” this nuance is still something to take into account when predicting the possible speedup in user applications.

60.11 Modifications and Further Optimizations

To minimize the number of memory accesses during a query, we can increase the block size. To find the local lower bound in a 32-element node (spanning two cache lines and four AVX2 registers), we can use a similar trick²⁶ that uses two `packs_epi32` and one `packs_epi16` to combine masks.

We can also try to use the cache more efficiently by controlling where each tree layer is stored in the cache hierarchy. We can do that by prefetching nodes to a specific level²⁷ and using non-temporal reads²⁸ during queries.

I implemented two versions of these optimizations: the one with a block size of 32 and the one where the last read is non-temporal. They don't improve the throughput:

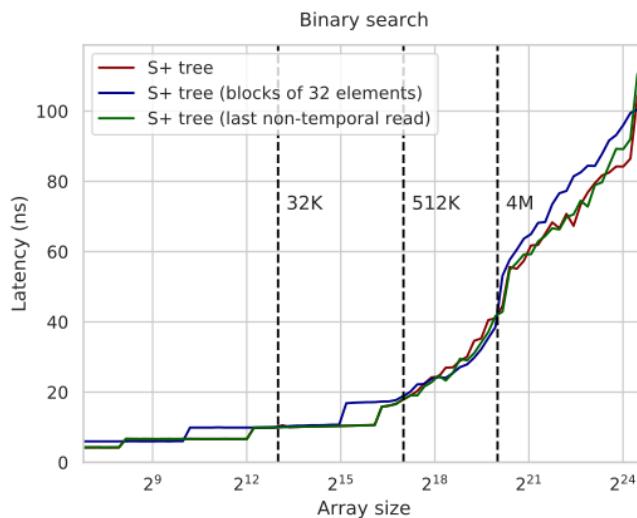


图 60.10: search-bplus-other

...but they do make the latency lower:

Ideas that I have not yet managed to implement but consider highly perspective are:

- Make the block size non-uniform. The motivation is that the slowdown from having one 32-element layer is less than from having two separate layers. Also, the root is often not full, so perhaps sometimes it should have only 8 keys or even just one key. Picking the optimal layer configuration for a given array size should remove the spikes from the relative speedup graph and make it look more like its upper envelope.
- I know how to do it with code generation, but I went for a generic solution and tried to implement²⁹ it with the facilities of modern C++, but the compiler can't produce optimal code this way.
- Group nodes with one or two generations of its descendants (300 nodes / 5k keys) so that they are close in memory—in the spirit of what FAST³⁰ calls hierarchical blocking. This reduces the severity

²⁶<https://github.com/sslotin/amh-code/blob/a74495a2c19dddc697f94221629c38fee09fa5ee/binsearch/bplus32.cc#L94>

²⁷<https://en.algorithmica.org/hpc/cpu-cache/prefetching/#software-prefetching>

²⁸<https://en.algorithmica.org/hpc/cpu-cache/bandwidth/#bypassing-the-cache>

²⁹<https://github.com/sslotin/amh-code/blob/main/binsearch/bplus-adaptive.cc>

³⁰http://kaldewey.com/pubs/FAST_SIGMOD10.pdf

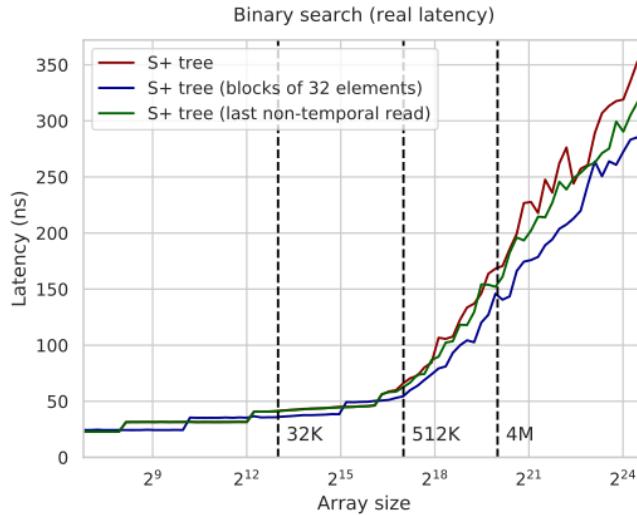


图 60.11: search-latency-bplus

of TLB misses and also may improve the latency as the memory controller may choose to keep the RAM row buffer³¹ open, anticipating local reads.

Optionally use prefetching on some specific layers. Aside from the $\frac{1}{17}$ -th chance of it fetching the node we need, the hardware prefetcher may also get some of its neighbors for us if the data bus is not busy. It also has the same TLB and row buffer effects as with blocking.

Other possible minor optimizations include:

- Permuting the nodes of the last layer as well —if we only need the index and not the value.
- Reversing the order in which the layers are stored to left-to-right so that the first few layers are on the same page.
- Rewriting the whole thing in assembly, as the compiler seems to struggle with pointer arithmetic.
- Using blending³² instead of **packs**: you can odd-even shuffle node keys ([1 3 5 7] [2 4 6 8]), compare against the search key, and then blend the low 16 bits of the first register mask with the high 16 bits of the second. Blending is slightly faster on many architectures, and it may also help to alternate between packing and blending as they use different subsets of ports. (Thanks to Const-me from HackerNews for suggesting³³ it.)
- Using **popcount**³⁴ instead of **tzcnt**: the index **i** is equal to the number of keys less than **x**, so we can compare **x** against all keys, combine the vector mask any way we want, call **maskmov**, and then calculate the number of set bits with **popcnt**. This removes the need to store the keys in any particular order, which lets us skip the permutation step and also use this procedure on the last layer as well.
- Defining the key *i* as the *maximum* key in the subtree of child *i* instead of the *minimum* key in the subtree of child (*i* + 1). The correctness doesn't change, but this guarantees that the result will be

³¹<https://en.algorithmica.org/hpc/cpu-cache-aos-soa/#ram-specific-timings>

³²<https://en.algorithmica.org/hpc/simd/masking>

³³<https://news.ycombinator.com/item?id=30381912>

³⁴<https://en.algorithmica.org/hpc/simd/shuffling/#shuffles-and-popcount>

stored in the last node we access (and not in the first element of the next neighbor node), which lets us fetch slightly fewer cache lines.

Note that the current implementation is specific to AVX2 and may require some non-trivial changes to adapt to other platforms. It would be interesting to port it for Intel CPUs with AVX-512 and Arm CPUs with 128-bit NEON, which may require some trickery³⁵ to work.

With these optimizations implemented, I wouldn't be surprised to see another 10-30% improvement and over 10x speedup over `std::lower_bound` on large arrays for some platforms.

60.12 As a Dynamic Tree

The comparison is even more favorable against `std::set` and other pointer-based trees. In our benchmark, we add the same elements (without measuring the time it takes to add them) and use the same lower bound queries, and the S+ tree is up to 30x faster:

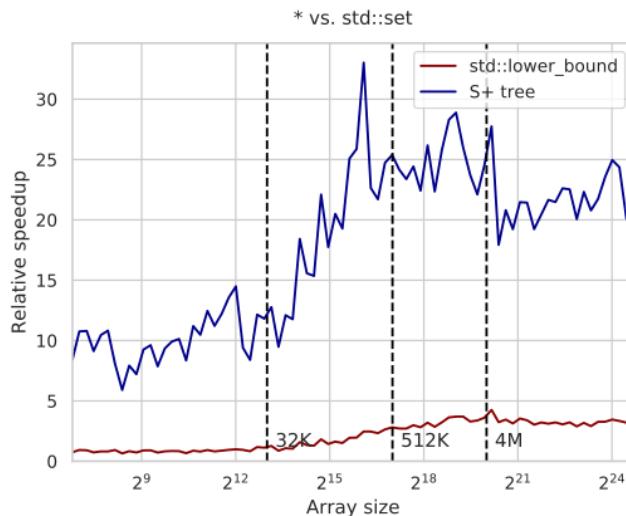


图 60.12: search-set-relative

This suggests that we can probably use this approach to also improve on *dynamic* search trees by a large margin.

To validate this hypothesis, I added an array of 17 indices for each node that point to where their children should be and used this array to descend the tree instead of the usual implicit numbering. This array is separate from the tree, not aligned, and isn't even on a hugepage —the only optimization we do is prefetch the first and the last pointer of a node.

I also added B-tree from Abseil³⁶ to the comparison, which is the only widely-used B-tree implementation I know of. It performs just slightly better than `std::lower_bound`, while the S+ tree with pointers is 15x faster for large arrays:

Of course, this comparison is not fair, as implementing a dynamic search tree is a more high-dimensional problem.

³⁵<https://github.com/WebAssembly/simd/issues/131>

³⁶<https://abseil.io/blog/20190812-btree>

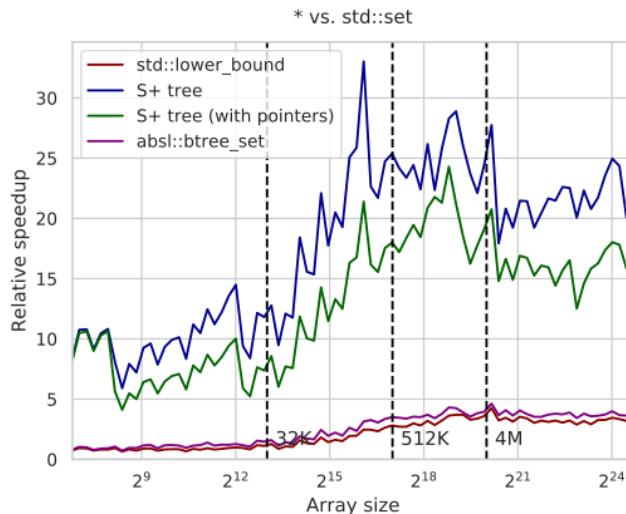


图 60.13: search-set-relative-all

We'd also need to implement the update operation, which will not be that efficient, and for which we'd need to sacrifice the fanout factor. But it still seems possible to implement a 10-20x faster `std::set` and a 3-5x faster `absl::btree_set`, depending on how you define “faster” —and this is one of the things we'll attempt to do next³⁷.

60.13 Acknowledgements

This StackOverflow answer³⁸ by Cory Nelson is where I took the permuted 16-element search trick from.

³⁷<https://en.algorithmica.org/hpc/data-structures/b-tree>

³⁸<https://stackoverflow.com/questions/20616605/using-simd-avx-sse-for-tree-traversal>

Chapter 61

Technique: Recursive variants and boxes

👤 Jonathan 📅 2022-05-31 💬 ★★★

There are many data structures that can be elegantly expressed using sum types. In C++ a (somewhat clunky) implementation of sum types is `std::variant`. However, it can't handle recursive data structures, where one alternative contains the entire sum type again.

Let's see how we can fix that.

61.1 The problem

We'll consider a simple calculator that supports addition and multiplication. We want to store and evaluate expressions like `11`, `40 + 2`, or `3 * 13 + 3`. That is, an expression is either a literal number, an addition containing two subexpressions, or a multiplication containing two subexpressions. Using `std::variant`, it can look like this:

```
struct LiteralExpr
{
    int value;
};

struct AddExpr
{
    Expr lhs, rhs;
};

struct MulExpr
{
    Expr lhs, rhs;
};
```

```
using Expr = std::variant<LiteralExpr, AddExpr, MulExpr>;
```

But of course, this doesn't compile: C++ requires a declaration before `Expr` can be used in `AddExpr`, but the declaration of `Expr` requires a declaration of `AddExpr`. Such circular dependencies can be solved by forward declaring `AddExpr` and `MulExpr` and moving the `Expr` declaration before their definition.

```
struct LiteralExpr
{
    int value;
};

// We forward declare the types while naming them here.
using Expr = std::variant<LiteralExpr,
                           struct AddExpr, struct MulExpr>

struct AddExpr
{
    Expr lhs, rhs;
};

struct MulExpr
{
    Expr lhs, rhs;
};
```

Instead of writing `struct foo;` `void f(const foo&);` you can combine the forward declaration and function declaration into one entity: `void f(const struct foo&);`. This is used in the declaration of `Expr` above.

Now, an expression like `1 + 2 * 3` would be stored as:

```
auto expr = Expr(AddExpr{LiteralExpr{1}, MulExpr{LiteralExpr{2}, LiteralExpr{3}}});
```

However, it still doesn't compile: `std::variant` does not work with forward declarations – it needs to know the size of the type, which requires a definition. And even if C++ were a language where declaration order does not matter, the circular dependency is still there.

Consider: what is the size of `Expr`?

Well, `Expr` is a variant, so its size is the size of the biggest member plus a tag. The biggest member is `AddExpr`, whose size is `2 * sizeof(Expr)`, which in turn may contain an `AddExpr`, whose size is `2 * sizeof(Expr)`, and so on. The only solution of `sizeof(Expr) = sizeof(tag) + 2 * sizeof(Expr)` is `sizeof(Expr) = 0` (or `sizeof(tag) = -sizeof(Expr)`)!

This is impossible.

61.2 Heap allocating nested expressions

One way to solve the infinite nesting is to only store e.g. an `AddExpr` if we actually need to store one, and leave it empty otherwise. This can be done by allocating an `AddExpr` on the heap whenever necessary.

That way, the variant itself only stores a pointer, which has a fixed size.

Since we're using modern C++, this means wrapping `AddExpr` and `MulExpr` inside `std::unique_ptr`:

```
using Expr = std::variant<LiteralExpr, std::unique_ptr<struct AddExpr>,
                           std::unique_ptr<struct MulExpr>>;
```

`std::unique_ptr` has no problems with forward declared types and is itself a complete type, so `std::variant` is happy. Instead of providing storage for infinite nesting, only as much memory is allocated as actually needed for a particular expression.

Instead of changing `Expr` to be a variant of `std::unique_ptr`, we could also change `AddExpr` and `MulExpr` to store `std::unique_ptr<Expr>` instead of `Expr`. However, this has two problems:

- It requires a forward declaration of the typedef `Expr`, which is not possible. We have to turn `Expr` into a struct that contains/inherits the variant.
- Instead of one allocation for each `AddExpr`, we now have two allocations, one for each operand of `AddExpr`.

This solution works.

It's also really ugly.

For starters, creating an expression requires `std::make_unique` calls:

```
Expr(std::make_unique<AddExpr>(LiteralExpr{1},
                                std::make_unique<MulExpr>(LiteralExpr{2}, LiteralExpr{3})));
```

And even that only works in C++20, where aggregates can be initialized with `T(args...)`. Otherwise, we need to add a constructor to `AddExpr` and `MulExpr`.

More importantly, `Expr` no longer has value semantics. Previously, we could freely copy `Exprs` which results in two independent objects (so no, `std::shared_ptr` isn't the answer). Now, thanks to `std::unique_ptr`, it is no longer copyable:

```
Expr square(Expr operand)
{
    // error: can't copy Expr
    return std::make_unique<MulExpr>(operand, operand);
}
```

Similarly, constness no longer propagates: when we have a `const Expr&` we could still modify lhs or rhs of an `AddExpr` as a `const std::unique_ptr<Expr>` still gives you an `Expr&`:

```
int evaluate(const Expr& expr)
{
    struct visitor
    {
        int operator()(const LiteralExpr& expr) { return expr.value; }
```

```

int operator()(const std::unique_ptr<AddExpr>& expr)
{
    expr->lhs = LiteralExpr{42}; // ups

    auto lhs = std::visit(*this, expr->lhs);
    auto rhs = std::visit(*this, expr->rhs);
    return lhs + rhs;
}

int operator()(const std::unique_ptr<MulExpr>& expr)
{
    auto lhs = std::visit(*this, expr->lhs);
    auto rhs = std::visit(*this, expr->rhs);
    return lhs * rhs;
}

return std::visit(visitor{}, expr);
}

```

Let's fix those problems.

61.3 Adding value semantics

In C++, we no longer use `malloc` ‘ed `const char*` pointers for string, where copying the pointer does not copy the string, we use `std::string`: it is the same internally, but adds value semantics on top. For the same reason, we should not use `std::unique_ptr`: it is only marginally better than raw pointers in that it provides and communicates ownership, but is fundamentally still a type with reference semantics. The only acceptable use of `std::unique_ptr` is as an implementation detail; it shouldn't appear in interfaces.

What we really want is a type that can store a heap allocated `T` but otherwise behaves like `T`. In particular, it should propagate `const`, and has a copy constructor that does a deep copy. Taking inspiration from Rust¹, let's call it `box<T>`:

```

template <typename T>
class box
{
    // Wrapper over unique_ptr.
    std::unique_ptr<T> _impl;

public:
    // Automatic construction from a `T`, not a `T*`.
    box(T &&obj) : _impl(new T(std::move(obj))) {}
    box(const T &obj) : _impl(new T(obj)) {}

```

¹<https://doc.rust-lang.org/std/boxed/index.html>

```

// Copy constructor copies `T`.
box(const box &other) : box(*other._impl) {}

box &operator=(const box &other)
{
    *_impl = *other._impl;
    return *this;
}

// unique_ptr destroys `T` for us.
~box() = default;

// Access propagates constness.
T &operator*() { return *_impl; }
const T &operator*() const { return *_impl; }

T *operator->() { return _impl.get(); }
const T *operator->() const { return _impl.get(); }
};

```

A couple things of note:

- It is a wrapper over `std::unique_ptr`. That way, we don't need to worry about the destructor.
- It can be implicitly constructed from `T`, which involves a heap allocation. This is similar to `std::string`, which can be implicitly constructed from `const char*`. For efficiency reason, the constructor can be made explicit, but this makes our intended usage with `std::variant` a bit more awkward.
- The copy constructor goes ahead and copies the `T` object, which requires allocating a new one. This is required for value semantics.
- Access to the underlying `T` object is possible using `operator*` and `operator->`. They propagate `const`: a `const box<T>` does only hand out `const T&`, unlike `std::unique_ptr`. In an ideal world, we had some sort of automatic dereferencing here to allow access with `..`, like Rust does.

Now we simply replace `std::unique_ptr` with `box` in the variant declaration. This makes construction nice again, we can freely copy expressions, and constness propagates.

```

using Expr = std::variant<LiteralExpr,
                           box<struct AddExpr>, box<struct MulExpr>>;
...
```

`auto expr = Expr(AddExpr{LiteralExpr{1}, MulExpr{LiteralExpr{2}, LiteralExpr{3}}});`

```

Expr square(Expr operand)
{

```

```

    return MulExpr{operand, operand}; // ok
}

int evaluate(const Expr& expr)
{
    struct visitor
    {
        int operator()(const LiteralExpr& expr) { return expr.value; }

        int operator()(const box<AddExpr>& expr)
        {
            // expr->lhs = LiteralExpr{42}; -- won't compile

            auto lhs = std::visit(*this, expr->lhs);
            auto rhs = std::visit(*this, expr->rhs);
            return lhs + rhs;
        }

        int operator()(const box<MulExpr>& expr)
        {
            auto lhs = std::visit(*this, expr->lhs);
            auto rhs = std::visit(*this, expr->rhs);
            return lhs * rhs;
        }
    };

    return std::visit(visitor{}, expr);
}

```

61.4 Aside: Moving boxes

Notice how I haven't given `box<T>` a move constructor. This is intentional, as there are two options and thus warrants more discussion.

The first is to have a move constructor that behaves like the copy constructor and moves the underlying `T` object. This requires heap allocating a new object, and makes it not `noexcept`:

```

box(box &&other) : box(std::move(*other._impl)) {}
box &operator=(box &&other)
{
    *_impl = std::move(*other._impl);
    return *this;
}

```

The second option is to delegate to `std::unique_ptr`'s move constructor, which transfers ownership.

This does not require heap allocation and makes it noexcept.

```
box(box&& other) noexcept = default;  
box& operator(box&& other) noexcept = default;
```

However, going with the second option introduces the possibility for a `box<T>` to be empty –the moved-from state. There, it is no longer allowed to access the underlying `T` object, as there is none.

As I've repeatedly² argued³ in the past, adding such a moved-from state is problematic, as the C++ compiler does not help you to catch it. If you go down that route, you should fully embrace the empty state –adding a default constructor, a query for it, etc. –turning the box into an `optional_box<T>`. Again, Rust doesn't have that problem as the compiler prevents access to moved objects.

61.5 Conclusion

Recursive variants require heap allocation; there is no way around that.

The simple approach to heap allocation is `std::unique_ptr`. However, it is a type with reference semantics, which are vastly inferior to value types. A better alternative is to write a simple wrapper over it that adds correct value semantics, `box<T>`.

In general, I don't really like `std::unique_ptr` for that reason. It has no place in interfaces and should only be an implementation detail. Unfortunately, the C++ standard library does not provide the nicer types, such as `box<T>` or the proposed `std::polymorphic_value<T>`, which is a replacement for polymorphic types⁴. This lead to a proliferation of reference semantics in interfaces, which is a shame.

²<https://www.foonathan.net/2016/07/move-safety/>

³<https://www.foonathan.net/2016/08/move-default-ctor/>

⁴<https://www.foonathan.net/2020/01/type-erasure/>

Chapter 62

Working with Strings in Embedded C++

👤 Niall Cooling 📅 2022-02-04 💬 ★★

In this post, by Embedded I'm generally referring to deeply embedded/bare-metal systems as opposed to Linux-based embedded systems.

62.1 Embedded systems and strings

Historically, the need for and thus the use of strings in embedded systems was fairly limited. However, this has changed with the advent of cheaper, full graphic displays and the growth of the ‘Internet of Things’ (IoT).

Many embedded systems sport full-colour graphics displays, supported by embedded-specific graphics libraries, including:

- free open-source –e.g. LVGL¹
- vendor-specific –e.g. TouchGFX² from STMicroelectronics
- fully specialised graphics environments –e.g. Qt for MCUs³.

Naturally, these environments will use strings extensively for labels, message boxes, alerts, etc.

Many of the major IoT frameworks utilise web services built on top of HTTP⁴, such as REST⁵. In conjunction with the web services, embedded applications utilise data interchange formats such as XML XCAP⁶ or JSON⁷. Both XML and JSON require character encoding based on ISO/IEC 10646⁸ such as UTF-8⁹.

¹<https://lvgl.io/>

²<https://www.st.com/en/development-tools/touchgfxdesigner.html>

³<https://doc.qt.io/QtForMCUs/>

⁴https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

⁵<https://restfulapi.net/>

⁶<https://datatracker.ietf.org/doc/html/rfc4825>

⁷<https://datatracker.ietf.org/doc/html/rfc7159>

⁸<https://www.iso.org/standard/76835.html>

⁹<https://datatracker.ietf.org/doc/html/rfc3629>

62.1.1 Character literals

Modern C++ extends the character literal model to support ISO 10646 character encoding. C++11¹⁰ (and C11) added UTF-16¹¹ and UTF-32¹² support, with C++20¹³ finally adding UTF-8 support.

```
int main()
{
    char      c1{ 'a' };           // 'narrow' char
    char8_t   c2{ u8'a' };        // UTF-8 - (C++20)
    char16_t  c3{ u'貓' };        // UTF-16 - (C11/C++11)
    char32_t  c4{ U' ' };         // UTF-32 - (C11/C++11)
    wchar_t   c5{ L' ' };         // wide char - wchar_t
}
```

Example Code¹⁴

62.2 C Strings

62.2.1 Null-Terminated Byte Strings (NTBS)

A ‘C-Style’ string is any **null-terminated byte string (NTBS)**, where this is a sequence of nonzero bytes followed by a byte with zero (0) value (the terminating null¹⁵ character). The terminating null character is represented as the character literal '\0';

The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.

A string literal (constant) is a sequence of characters surrounded by double quotes ("").

```
#include <cstring>
#include <iostream>

int main(void)
{
    char message[] = "Hello World";

    std::cout << sizeof(message) << '\n'; // 12
    std::cout << strlen(message) << '\n'; // 11
}
```

Example Code¹⁶

In C/C++, single quotes (') are used to identify character literals. Single quotes (') cannot be used (unlike some other programming languages) to represent strings.

¹⁰<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2249.html>

¹¹<https://www.rfc-editor.org/rfc/rfc2781.html>

¹²<https://en.wikipedia.org/wiki/UTF-32>

¹³<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0482r6.html>

¹⁴<https://godbolt.org/z/-3PGNW>

¹⁵<http://eel.is/c++draft/lex.charset#tab:lex.charset.literal>

¹⁶<https://godbolt.org/z/dqnrKb>

62.2.2 C-Strings and string literals

What is the difference in the memory model between the following two program object definitions?

```
#include <iostream>

int main()
{
    char message[] = "this is a string";
    std::cout << sizeof(message) << '\n';

    const char *msg_ptr = "this is a string";
    std::cout << sizeof(msg_ptr) << '\n';
}
```

Example Code¹⁷

The first output message will display 17, the number of characters in the string (including the null character). The second output will display the sizeof a pointer (e.g. 4 on 32-bit Armv7-M).

Although the code above looks ostensibly identical, there are significant semantic differences:

- For `message`, the memory for the array is allocated on the stack at runtime. The compiler initialises it from the string literal. At runtime, the program memory copies the string literal into the array (depending on compiler optimisations and ISA¹⁸).
- For `msg_ptr`, only the address of the string literal is held on the stack, and there is no copying of string literal.

String literals are stored in your program image, usually in a read-only section (.rodata), normally mapped to NVM¹⁹ such as Flash²⁰.

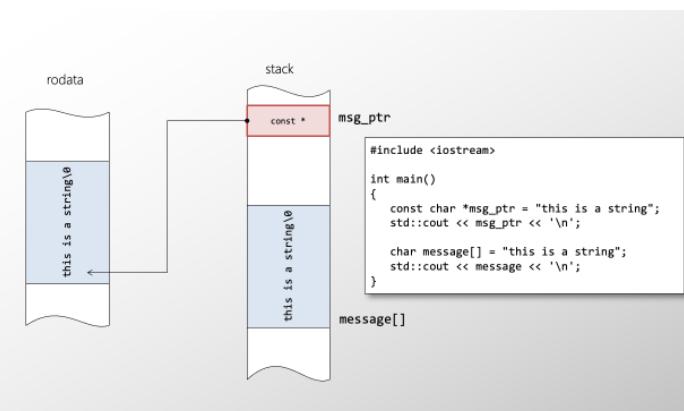


图 62.1: text

As the memory for `message` is Stack-based, at runtime, it is allowable to manipulate the contents of `message[]` using indexing, e.g.

¹⁷<https://godbolt.org/z/7KeKa66qM>

¹⁸<https://www.arm.com/glossary/isa>

¹⁹https://en.wikipedia.org/wiki/Non-volatile_memory

²⁰https://en.wikipedia.org/wiki/Flash_memory

```
message[0] = 'T';
```

Unlike C, C++ does not allow the dangerous code of having a non-const pointer pointing at constant memory, e.g.:

```
char *msg_ptr = "this is a string";
```

as this would allow the statement:

```
msg_ptr[0] = 'T';
```

which on most systems will cause a program failure by trying to write to readonly memory.

62.2.3 Finding strings

Most modern C/C++ toolchains (e.g. GNU Arm Embedded Toolchain²¹) supply a collection of useful utilities, usually referred to as Binutils²².

The strings²³ utility lists printable strings from a file. Running strings on an application image will list all embedded strings. This is a common tool for bad actors to look for embedded clear-text passwords, etc., in firmware (e.g. Intro To Hardware Hacking -Dumping Your First Firmware²⁴).

```
$ arm-none-eabi-strings -d Application.elf | grep "this is"
this is a string
```

Strings can also be run against an individual object file, e.g.

```
$ arm-none-eabi-strings -d -n 12 main.o
this is a string
```

It can be enlightening to run strings against your own application image.

Another utility, objdump²⁵, can help us understand the actual memory image. Running objdump against the object file can identify whether the code uses read-only data, e.g.

```
$ arm-none-eabi-objdump -h main.o | grep rodata
77 .rodata.main.str1.4 00000011 00000000 00000000 000003d0 2**2
```

Without getting into the details of objdump, this output tells us that the main object file has *readonly* data of size **0x11** (17 bytes). The symbol name is compiler generated.

At the link stage, we generate a **.map** file (you should be if you’re not already, it contains a wealth of target information). Search the map file for the symbol **.rodata.main.str1.4** from objdump; we can see it is located at the address **0x08023b70**. On our Cortex-M4 target, this is on-chip Flash.

```
.rodata.main.str1.4
0x0000000008023b70      0x11 main.o
```

²¹<https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm>

²²<https://www.gnu.org/software/binutils/>

²³<https://man7.org/linux/man-pages/man1/strings.1.html>

²⁴<https://blog.nvisium.com/intro-to-hardware-hacking-dumping-your-first-firmware>

²⁵<https://www.man7.org/linux/man-pages/man1/objdump.1.html>

62.2.4 C String Standard Library

Standard C supplies a library to help with common string manipulation issues (`<string.h>` in C or `<cstring>` in C++). It has several ‘helper’ functions to manipulate NTBS, e.g.

copying strings	<code>strcpy</code> , <code>strncpy</code>
concatenating strings	<code>strcat</code> , <code>strncat</code>
comparing strings	<code>strcmp</code> , <code>strncmp</code>
parsing strings	<code>strtok</code> , <code>strcspn</code>
length	<code>strlen</code>

Note that all these functions rely on the supplied pointer pointing at a well-formed NTBS. The behaviour is undefined if it is not a pointer to a null-terminated byte string.

62.2.5 Safety and Security Issues

Unfortunately, using strings in embedded systems potentially introduces several safety and security weaknesses. SEI CERT C²⁶ has some rules regarding the misuse of strings, and the Common Weakness Enumeration²⁷ website covers many potential types of string-related weaknesses. Many of these arise from aspects such as incorrect string termination, memory buffer errors and incorrect data validation (string parsing).

62.3 C++ Strings

62.3.1 C++11 Raw String Literals

C++11 introduced a Raw string literal²⁸ type. In a raw string, literal escape sequences are not processed; and the resulting output is exactly the same as appears in the source code. Raw string literals are useful for storing file paths or regular expressions, which use characters that C++ may interpret as formatting information. They can be represented in UTF format as needed.

For example, given the following code:

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

const char* raw_str { // could use constexpr auto raw_str
R"(<!DOCTYPE html>
<html>
<body>
    <p>hello, world!</p>
</body>
</html>
)"}
```

²⁶<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152038>

²⁷<https://cwe.mitre.org/data/definitions/699.html>

²⁸<http://eel.is/c++draft/lex.string#:raw-string>

```

};

int main()
{
    std::cout << raw_str;
}

```

The standard output is:

```

<!DOCTYPE html>
<html>
<body>
    <p>hello, world!</p>
</body>
</html>

```

The raw program image will contain (depending on ISA) the full string, e.g. Intel

```
.string "<!DOCTYPE html>\n<html>\n<body>\n <p>hello, world!</p>\n</body>\n</html>\n"
```

Arm:

```

.LC0:
.ascii "<!DOCTYPE html>\012<html>\012<body>\012 <p>hello"
.ascii ", world!</p>\012</body>\012</html>\012\000"

```

Example Code²⁹

62.3.2 C++ Strings Library

The C++ Standard Library supports the header `<string>`. The basic `std::string` can be thought of as a variable-sized character array (a vector of characters). `std::string` supports operator overloads for the most common string behaviour – converting from string literals, concatenation, etc. making string parsing more straightforward than using NTBSs. `std::string` is also supported by `<iostream>`(e.g. `std::cout / std::cin`), giving one significant benefit over C, in that an input string will grow to the size of the input stream (i.e. no pre-allocation of a local array of chars).

```

#include <iostream>
#include <string>

const std::string salutation { "Hello: " };           // initialise

int main()
{
    std::cout << "Enter your name: ";

```

²⁹<https://godbolt.org/z/Pn8x4x>

```

std::string name{};
std::cin >> name;                                // reads to whitespace
// std::getline(std::cin, name);                  // reads to '\n'

std::string greeting = salutation + name + '\n'; // concatenation

std::cout << greeting;

if(not salutation.empty()) {
    std::cout << salutation.length() << '\n';
    std::cout << salutation[0] << ' ' << salutation.front() << '\n';
    std::cout << salutation[salutation.length()-1] << ' ' << salutation.back() << '\n';
}
}

```

62.3.3 Numeric to std::to_string

C++11 introduced the overloaded function `std::to_string`. This, unsurprisingly, converts a numeric value to `std::string`. E.g.

```

#include <iostream>
#include <string>

int main()
{
    int i = 42;
    unsigned long ul = 42UL;
    auto i_str = std::to_string(i);
    auto ul_str = std::to_string(ul);
    std::cout << "std::cout: " << i << '\n'
        << "to_string: " << i_str << "\n"
        << "std::cout: " << ul << '\n'
        << "to_string: " << ul_str << "\n";
    printf("%i\n%s\n%li\n%s\n", i, i_str.c_str(), ul, ul_str.c_str());

    double f = 23.43;
    double f2 = 1e-9;
    auto f_str = std::to_string(f);
    auto f_str2 = std::to_string(f2); // Note: returns "0.000000"

    std::cout << "std::cout: " << f << '\n'
        << "to_string: " << f_str << "\n"
        << "std::cout: " << f2 << '\n'
        << "to_string: " << f_str2 << "\n";
}

```

```

    printf("%g\n%s\n%g\n%s\n", f, f_str.c_str(), f2, f_str2.c_str());
}

```

Example Code³⁰

Expected output

```

std::cout: 42
to_string: 42
std::cout: 42
to_string: 42
42
42
42
42
std::cout: 23.43
to_string: 23.430000
std::cout: 1e-09
to_string: 0.000000
23.43
23.430000
1e-09
0.000000

```

62.3.4 std::string and NTBS

`std::string` cannot be used directly where a `const char*` is required. Many embedded libraries favour C-based APIs, requiring support for converting a `std::string` to `const char*`.

`std::string` has a member function `.c_str()`, which returns a pointer to the underlying C-style string (`const char*`). In addition, `std::string`, as with most container types, can get at the underlying raw data via a member function, `.data()`, or the standard library function `std::data()`.

```

#include <iostream>
#include <string>
#include <cstdio>

int main()
{
    const char *str = "Hello World!";

    std::string s { str };

    std::cout << "Using cout: " << s << '\n';

    std::printf("C-style string: %s \n", str);

```

³⁰<https://godbolt.org/z/3YY9qcYP9>

```

    std::printf("std::string as C-style string: %s \n", s.c_str() );
    std::printf("std::string as raw data: %s \n", s.data() );
}

```

For both models, the pointer returned is such that the range `[data(); data() + size()]` is valid and the values in it correspond to the values stored in the string. Therefore `data() + i == std::addressof(operator[](i))` for every `i` in `[0, size()]` is guaranteed.

Note, there was a minor change to `.data()` in C++17. Prior to C++17, `.data()`, like `.c_str()` returned `const char*`. Since C++17, `.data()` now returns `char*`.

When writing library code is recommended never to let C++ strings propagate outside your component. Different compilers have different models for strings; for portability prefer `const char*` parameters.

62.3.5 typedefs for string types

As the character literal model was extended to support ISO 10646 character encoding, the standard library has string support for each character type, i.e.

Type	Definition
<code>std::string</code>	<code>std::basic_string<char></code>
<code>std::wstring</code>	<code>std::basic_string<wchar_t></code>
<code>std::u8string</code> (C++20)	<code>std::basic_string<char8_t></code>
<code>std::u16string</code> (C++11)	<code>std::basic_string<char16_t></code>
<code>std::u32string</code> (C++11)	<code>std::basic_string<char32_t></code>

These are all based on a common underlying template class of:

```

template<class CharT,
         class Traits = std::char_traits<CharT>,
         class Allocator = std::allocator<CharT>
>
class basic_string;

```

62.4 String memory management

62.4.1 Strings are, by default, heap-allocated

By default, `std::string` uses `std::allocate`, which in turn, uses `::new/::delete` to allocate dynamic memory for storing the actual NTBS.

```

#include <string>

int main() {
    const char* s1 = "literal string"; // characters stored in .rodata
    std::string s2 = "Literal String"; // characters stored in .heap
}

std::string have three main parts:

```

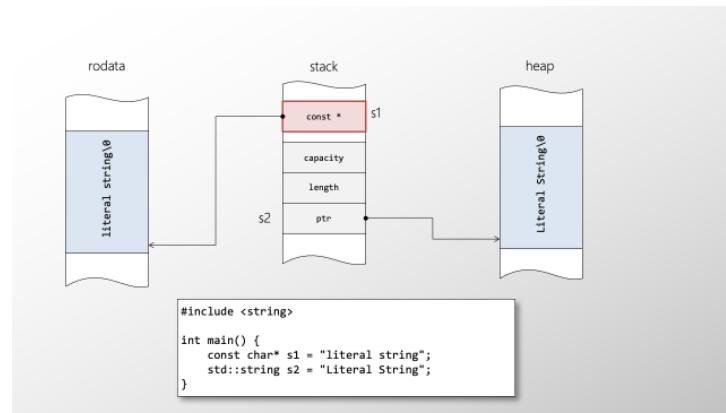


图 62.2: text

- data -> stored on the heap; accessed via .data()
 - length -> based on strlen; accessed via .length() or .size()
 - capacity -> allocated size on the heap (\geq length); accessed via .capacity()
- Capacity can be extended using .reserve() or reduced using .shrink_to_fit().

Note, looking at the definition of basic_string:

```
template<class CharT, ..., class Allocator = std::allocator<CharT>> class basic_string;
```

you may notice you can replace the use of ::new/:/delete with a different allocator (more on this later).

62.4.2 Copying Strings

A string's handle lifetime defines the lifetime of the underlying string, e.g.

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

std::string s1 { "initial contents" };

int main() {
    std::cout << s1 << '\n';
    {
        auto s2{ s1 };           // copy-construction
        s2[0] = 'I';            // unchecked assignment
        std::cout << s2 << '\n';
    }                         // s2 heap memory deleted
    std::cout << s1 << '\n';
    {
        std::string s3{};      // empty string
        s3 = s1;              // copy-assignment
        s3.at(0) = 'I';        // bound-safe assignment
        std::cout << s3 << '\n';
    }
}
```

```

    }                                // s3 heap memory deleted
    std::cout << s1 << '\n';
}

```

Expected Output

```

initial contents
Initial contents
initial contents
Initial contents
initial contents

```

For `s1`, the lifetime is `static`, but the memory for the NTBS is still dynamically allocated (i.e. `::new` is called before `main`). For `s2` and `s3`, `::new` is called when the copy takes place, and `::delete` is called when reaching the end of enclosing blocks.

Example Code³¹

62.4.3 Deep copy of std::string

`std::string` implements “deep-copy”³² semantics; when the copies are created, new memory is allocated, and the contents are copied (e.g. `strcpy`)

```

int main() {
    std::string s1 = "literal string";
    auto s2 { s1 }; // ::new called and characters copied
}

```

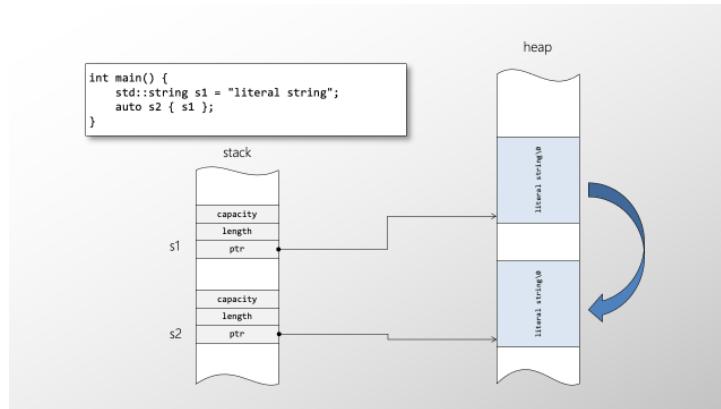


图 62.3: text

62.4.4 Moving std::string

The addition of Move-semantics for Modern C++ was driven by reducing deep-copying of complex objects, such as strings.

³¹<https://godbolt.org/z/qGz6er>

³²<https://blog.feabhas.com/2014/12/the-rule-of-the-big-three-and-a-half-resource-management-in-c/>

```
int main() {
    std::string s1 = "literal string";
    auto s2 = std::move(s1); // only the address of the .heap memory is copied
}
```

When `s1` is initially constructed, the memory model will look somewhat like this:

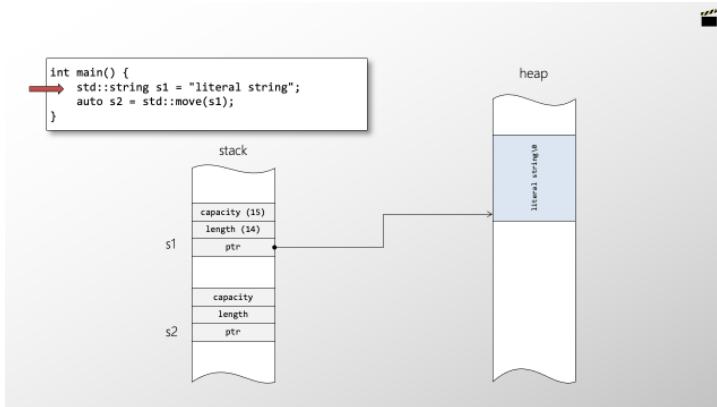


图 62.4: text

As before, the memory for the string handle, `s2`, is allocated on the stack. When a string is moved, rather than copied, the pointer to the heap-based NTBS is copied to the new object (`s2`), rather than the whole NTBS part being copied, e.g.

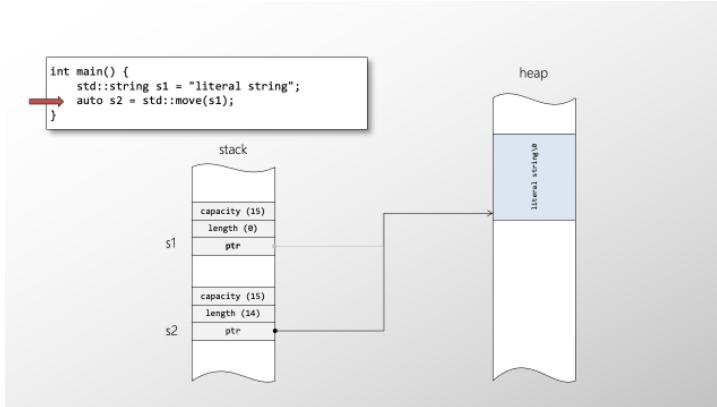


图 62.5: text

The length of the moved-from string (`s1`) becomes zero (0). Note, however, current implementations don't necessarily reset the moved-from pointer to the `nullptr` (0) as would be safe. The standard specifies the moved-from string "*...is left in a valid state with an unspecified value.*" .

62.4.5 Short String Optimisation (SSO)

Many modern compilers (e.g. GCC and clang) support an implementation-specific optimisation generally referred to as Short-String Optimisation (SSO). As we've seen, there is the handle part for every string. This stores the three essential data items:

- data

- size
- capacity

For small strings, this typically proves an overhead will typically outweigh, both from a performance and memory perspective, the benefits of using `std::string`.

So to significantly improve performance, when the literal string has fewer characters than an implementation-defined threshold, the runtime implementation stores the literal string within the stack space allocated for the handle, rather than allocation memory from the heap, e.g.

The screenshot shows the Godbolt compiler interface with the following code:

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

void* operator new(size_t sz) {
    std::cout << "[allocating " << sz << " bytes]\n";
    return std::malloc(sz);
}

int main() {
    std::string s1 { "0123456789" };
    std::string s2 { "01234567890123456789" };
}
```

A callout box points to the allocation of `s2` with the text "only dynamic memory allocation here". Below the code, a terminal window shows the output: "[allocating 21 bytes]".

图 62.6: text

Example Code³³

In the example shown, we can see from the output dynamic memory has only occurred for the larger of the two strings.

62.4.6 SSO -GCC

Different compilers have their own specific implementation of SSO. For example, compiled for GCC/Linux on an x86-64 architecture, the stack-based part of the string is 32-bytes (where the stack is 8-bytes wide). When using SSO, the upper 16-bytes are used to store the NTBS (thus placing a threshold of 16-bytes, including the terminating null character, for SSO).

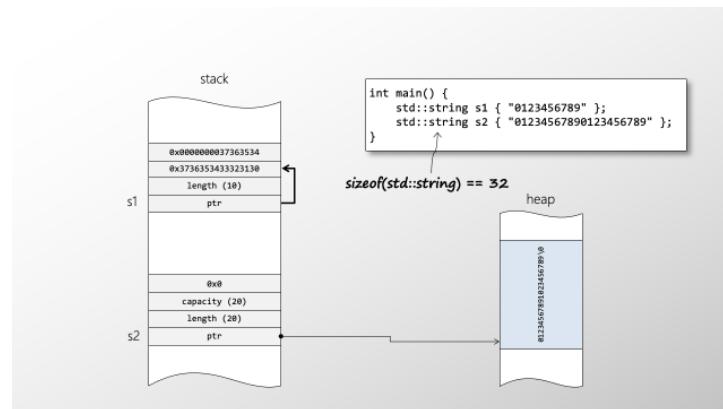


图 62.7: text

³³<https://godbolt.org/z/o9a8hv>

Example Code³⁴ implementation based on GCC v10.2

On a 32-bit platform, where the stack is 4-bytes wide, GCC uses a 24-bytes structure to implement SSO (`sizeof(std::string) == 24`). Again, the upper 16-bytes can store characters when utilising SSO in this model.

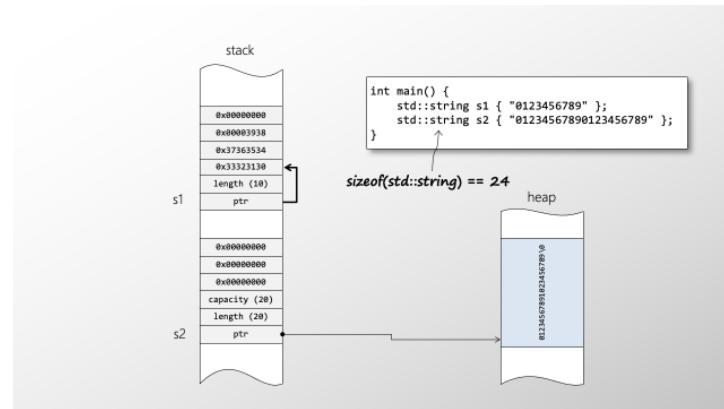


图 62.8: text

Based on: GCC version 9.2.1 (GNU Tools for Arm Embedded Processors 9-2019-q4-major).

Clang (on a 64-bit platform) uses a different implementation for SSO. When SSO is employed, the first byte stores the calculated string size. The last double-word always stores the value 33. This allows up to 22 characters to be held on the stack before the heap usage is required.

62.4.7 String APIs

An NTBS can be passed as an argument for the function parameter of type `const std::string&` (const lvalue reference). However, as part of the call, a temporary `std::string` object is created to bind to the parameter, and `::new` is called to generate the rvalue expression `std::string{s1}`.

```
#include <string>

void c_str(const char* str) {           // pass-by-pointer-to-const
    ...
}

void cpp_str(const std::string& str) {   // pass-by-const-lvalue-ref
    ...
}

int main() {
    const char* s1 { "godbolt compiler explorer" }; // NTBS
    std::string s2 { "godbolt compiler explorer" }; // Heap-based C++ String
    c_str(s1);
    c_str(s2);          // FAILS to compile
}
```

³⁴<https://godbolt.org/z/Y7z6oq>

```

c_str(s2.c_str()));

cpp_str(s1);           // cpp_str(std::string{s1});
cpp_str(s2);
}

```

As we have both NTBS and C++ Strings, writing portable code to handle both can be challenging. For example, a C++ string cannot be passed to `const char *` (the `.c_str()` member function must be used).

Example Code³⁵

62.5 C++17 std::string_view

C++17 introduced significant library support for string management in the form of `std::string_view`. `std::string_view` describes an object that can refer to a constant contiguous sequence of char-like objects, with the first element of the sequence at position zero.

This means `std::string_view` can handle both NTBS and `std::string`, e.g.

```

#include <string>
#include <string_view>

void cpp_sv(std::string_view str) {
    ...
}

int main()
{
    const char* s1 { "godbolt compiler explorer" }; // NTBS
    std::string s2 { "godbolt compiler explorer" }; // Heap-based C++ String

    cpp_sv(s1);
    cpp_sv(s2);
}

```

A typical implementation holds only two members: a pointer to constant character type (`CharT`) and a size.

The header `<string_view>` defines several typedefs for common character types, e.g.

Type Definition

<code>std::string_view</code>	<code>std::basic_string_view<char></code>
<code>std::wstring_view</code>	<code>std::basic_string_view<wchar_t></code>
<code>std::u8string_view</code> (C++20)	<code>std::basic_string_view<char8_t></code>
<code>std::u16string_view</code>	<code>std::basic_string_view<char16_t></code>
<code>std::u32string_view</code>	<code>std::basic_string_view<char32_t></code>

³⁵<https://godbolt.org/z/M3djMd>

62.5.1 Constant pointer + size

The common implementation of `std::string_view` is a class that holds a pointer-constant to the character array and its size, e.g.

```
template<class _CharT, class _Traits = char_traits<_CharT> >
class basic_string_view {
public:
    typedef _CharT value_type;
    typedef size_t size_type;
    ...
private:
    const value_type* __data;
    size_type __size;
};
```

Note that `__data` is `const*`.

So the effective calling code is:

```
int main()
{
    const char* s1 { "godbolt compiler explorer" };
    std::string s2 { "godbolt compiler explorer" };

    cpp_sv(s1); // string_view(s1, strlen(s1))
    cpp_sv(s2); // string_view(s2.data(), s2.length)
}
```

Example Code³⁶

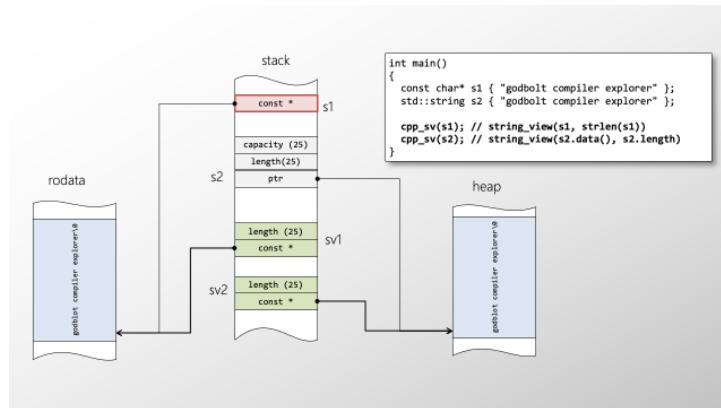


图 62.9: text

³⁶<https://godbolt.org/z/MPcMon>

62.5.2 std::string_view as a NTBS replacement

With the introduction of `std::string_view` in C++17, we now have a safer, lightweight alternative to using C-style NTBS e.g.

```
#include <iostream>
#include <string_view>

int main()
{
    const char*      s1 { "compiler explorer" };      // NTBS in .rodata
    std::string_view s2 { "Compiler Explorer" };        // also stored in .rodata

    std::cout << sizeof(s1) << '\n';                  // 4/8 - 32/64bit
    std::cout << sizeof(s2) << '\n';                  // 8/16

    std::cout << s2.length() << '\n';                // could also use .size()
    if(s2.empty()) std::cout << "empty string\n";
    if(s2.compare(s1) == 0) std::cout << "strings are equal\n";

    for(auto c : s2) {
        std::cout << c << '\n';
    }

    std::string_view s3 { "ALL CAPITALS" };
    std::swap(s2, s3);                                // also s2.swap(s3)
}
```

Example Code³⁷

The `sizeof(std::string_view)` is `sizeof(void*)+sizeof(std::size_t)` where `std::size_t` is typically `typedef`ed to unsigned long int.`

It is helpful that we can quickly wrap NTBSs in `string_view` objects to allow processing on them using standard library elements, e.g.

```
if (argc > 1) std::string_view run_count_str{ argv[1] };
```

62.5.3 Common APIs with string-like syntax

The primary benefit of using `std::string_view` is many string parsing functions are available through a common interface, e.g.

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

³⁷<https://godbolt.org/z/K11vrT>

```

void cpp_sv(std::string_view str) {
    std::cout << str.length() << ' ';
    std::cout << reinterpret_cast<const void*>(str.data()) << '\n';
    if(str == "godbolt compiler explorer"){
        std::cout << "strings equal\n";
    }
    if(not str.empty()){
        constexpr std::string_view compiler{"compiler"};
        std::cout << str.substr(str.find(compiler), compiler.length()) << '\n';
    }
}

int main() {
    const char* s1 { "godbolt compiler explorer" }; // NTBS
    std::string s2 { "godbolt compiler explorer" }; // C++ Heap string
    cpp_sv(s1); // string_view(s1, strlen(s1))
    cpp_sv(s2); // string_view(s2.data(), s2.length())
}

```

Example Code³⁸

Expected Output

```
[allocating 26 bytes]
25 0x40201a
strings equal
compiler
25 0x1cd3ec0
strings equal
compiler
```

Most `std::string_view` functions work identically to `std::string` but without the potential of generating temporary (rvalue expression) strings (e.g. `==`).

62.5.4 auto type deduction

When using auto type deduction, a quoted string defaults to `const char*`. The standard library supplies literals for both `std::string` and `std::string_view`. Appending 's' to a quoted-string converts a character array literal to `basic_string`, whereas appending 'sv' to a quoted-string creates a string view of a character array literal.

As well as auto type deduction, this also applies to template deduction, e.g.

```
#include <string>
#include <string_view>
#include <typeinfo>
```

³⁸<https://godbolt.org/z/5W4obK>

```
#include <cassert>

using namespace std::string_literals; // operator""s
using namespace std::literals; // operator""sv

int main() {
    const char* ntbs_1 { "godbolt compiler explorer" }; // NTBS
    std::string str_1 { "godbolt compiler explorer" }; // C++ Heap string
    std::string_view sv_1 { "godbolt compiler explorer" }; // std::string_view

    auto ntbs_2 { "godbolt compiler explorer" }; // const char *
    auto str_2 { "godbolt compiler explorer"s }; // std::string
    auto sv_2 { "godbolt compiler explorer"sv }; // std::string_view

    assert(typeid(ntbs_1) == typeid(ntbs_2));
    assert(typeid(str_1) == typeid(str_2));
    assert(typeid(sv_1) == typeid(sv_2));
}
```

Example Code³⁹

62.5.5 auto vs decltype

One area of modern C++ that may still trip you up is the subtle difference between `auto` and `decltype` for the NTBS strings. As previously mentioned, an NTBS string is automatically deduced as `const char*`, whereas when using `decltype` to inspect an entity or expression, an NTBS yields a character array based on `strlen+1`, e.g.

```
#include <typeinfo>
#include <iostream>

int main()
{
    const char* str { "hello" };
    auto s1 { "Hello" };
    decltype( "Hello" ) s2 {};
    auto s3 { str };
    decltype( str ) s4 {};

    std::cout << typeid(str).name() << '\n'; // const char*
    std::cout << typeid(s1).name() << '\n'; // const char*
    std::cout << typeid(s2).name() << '\n'; // char[6]
    std::cout << typeid(s3).name() << '\n'; // const char*
```

³⁹<https://godbolt.org/z/Gxbd1v>

```

    std::cout << typeid(s4).name() << '\n';           // const char*
    std::cout << typeid("hello").name() << '\n';     // char[6]
}

```

62.6 std::string_view Caveats

So we've seen that `std::string_view` is an ideal candidate for possibly refactoring legacy code (where appropriate) by replacing both `const char*` and `const std::string&` parameters with `std::string_view`. However, as with most things in life, there are always a couple of gotchas.

There are two significant uses where `std::string_view` can fail:

- lifetime management
- non-null-terminated strings

62.6.1 string lifetime management

First, it is the programmer's responsibility to ensure that `std::string_view` does not outlive the pointed-to character array. To get this wrong does take some effort, and in well-crafted code should never happen, but it is still worth being aware of, e.g.

```

#include <iostream>
#include <string>
#include <string_view>

using namespace std::string_literals; // operator""s
using namespace std::literals;       // operator""sv

int main()
{
    std::string_view ntbs{ "a string literal" };           // OK: points to a static array
    std::string_view heap_string{ "a temporary string"s }; // rvalue string using new
    // rvalue string memory deallocated...
    std::cout << "Address of heap_string: " << (void*)heap_string.data() << '\n';
    std::cout << "Data at heap_string: " << heap_string.data() << '\n';
}

```

Example Code⁴⁰

In this example, the string used to construct `heap_string` is an rvalue expression⁴¹; this will call `::new` during construction. However, rvalue objects lifetime only exists for the duration of the statement. Before we execute the following statement, `::delete` will be called as the rvalues object lifetime has ended.

Accessing an object after its lifetime has ended is undefined behaviour. In a host system (e.g. Linux), this error will be caught using one of the modern compiler sanitisers (e.g. ASan⁴²). In a target system, it is

⁴⁰<https://godbolt.org/z/c9Gx8f>

⁴¹https://en.cppreference.com/w/cpp/language/value_category

⁴²<https://clang.llvm.org/docs/AddressSanitizer.html>

unlikely to be caught at runtime and could lead to difficult to find bugs. Example with ASan enabled⁴³.

62.6.2 non null-terminated strings

Secondly, unlike `string::data()` and string literals, `string_view::data()` may return a pointer to a buffer that is not null-terminated (e.g. a substring). Therefore it is typically a mistake to pass `data()` to a function that takes just a `const charT*` and expects a null-terminated string. `std::string_view` is **not guaranteed** to be pointing at an NTBS, e.g.

```
#include <iostream>
#include <string>
#include <string_view>
#include <cstdio>

using namespace std::string_literals; // operator""s
using namespace std::literals;       // operator""sv

void sv_print(std::string_view str) {
    std::cout << str.length() << ' ' << reinterpret_cast<const void*>(str.data()) << '\n';
    std::cout << "cout: " << str << '\n';      // based on str.length()
    printf("stdout: %s\n", str.data());          // based on NUL
}

int main() {
    std::string      str_s {"godbolt compiler explorer"};
    std::string_view str_sv {"godbolt compiler explorer"};
    char char_arr2[] = {
        'a', ' ', 'c', 'h', 'a', 'r', ' ', 'a', 'r', 'r', 'a', 'y'
    }; // Not null character terminated
    sv_print(str_s.substr(8,8));
    sv_print(str_sv.substr(8,8));
    sv_print(char_arr2);
}
```

Example Code⁴⁴

Example Output

```
[allocating at 0x1454eb0 size: 26 bytes]
8 0x7fff14078d0
cout: compiler
stdout: compiler
8 0x402052
cout: compiler
```

⁴³<https://godbolt.org/z/TKEKz9f5a>

⁴⁴<https://godbolt.org/z/jaTGaa>

```
stdout: compiler explorer
16 0x7fffc14078e4
cout: a char array NE
stdout: a char array NE
[deallocating at 0x1454eb0]
```

Again, *your mileage may vary* depending on running this on a host versus a target system. The address sanitizer, ASan, will typically detect and report on this error.

62.7 Polymorphic Memory Resource (PMR) Strings

I want to finish by touching on an area that probably deserves its own article.

C++17 introduced library support for **Memory Resources**. Memory resources implement memory allocation strategies that can be used by `std::pmr::polymorphic_allocator`, covered in detail in a previous blog post⁴⁵.

Simply, *PMR* allows you to specify a chunk of allocated memory (typically a simple stack-based array) as the memory to be used instead of the heap. C++17 support PMR based strings, e.g.

```
#include <iomanip>
#include <iostream>
#include <new>
#include <string>

#include <memory_resource> // C++17 header

// print if ::new called
void* operator new(size_t sz) { ... }

// dump array contents in hex
auto print_mem = [] (auto& b) { ... };

// print capacity size string
auto print_string = [] (auto& s) { ... };

int main() {
    std::array<uint8_t, 32> buff{}; // stack based array

    // create PMR buffer
    std::pmr::monotonic_buffer_resource buffer_mem_res(
        buff.data(), buff.size(), std::pmr::null_memory_resource());
    print_mem(buff);

    std::pmr::string str("hello", &buffer_mem_res); // SSO still used
    print_string(str);
    print_mem(buff);
```

⁴⁵<https://blog.feabhas.com/2019/03/thanks-for-the-memory-allocator/>

```

str = "012345678901234567890";    // replace ::new with pmr::
print_string(str);
print_mem(buff);
str = "01234567890123456789012";
print_string(str);
print_mem(buff);
}

```

Example code⁴⁶

Expected Output

```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 5 hello
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
30 21 012345678901234567890
303132333435363738393031323334353637383930 0 0 0 0 0 0 0 0 0 0
30 23 01234567890123456789012
3031323334353637383930313233343536373839303132 0 0 0 0 0 0 0 0

```

The use of `std::pmr::null_memory_resource()` means that if the buffer request exceeds the buffer size, the program will terminate (either through raising the `std::bad_alloc` exception or calling `terminate`). You can also cascade PMR buffers (i.e. we could have a bigger array to use if the 32-byte limit is exceeded) or revert to using the standard heap.

Unfortunately, as of today, there is still minimal support for PMR across compiler toolchains.

62.8 Summary

With the growth of IoT and embedded graphics, the need to use strings has become more commonplace in modern, deeply embedded systems. C-style string management (NTBS) with the associated library is challenging compared to modern programming languages and opens up the potential introduction of program flaws.

The C++ standard library has always supported a more flexible and rich programming class for strings. Unfortunately, `std::string` is not appropriate for many deeply embedded systems due to its requirement for dynamic memory management.

C++17's introduction of `std::string_view` allows us to replace many uses of NTBS parsing with the more user-friendly `string_view` parsing functions.

SSO means, with care, that we can be safely use string management for small strings (e.g. fewer than 16 characters for GCC) in an embedded environment. However, implementing strategies to not spill onto the heap may outweigh the benefits.

`std::pmr::string` is a real potential to allow complete, modern string management to be used in embedded C++, bringing with it potential safety and security benefits over using NTBS. Unfortunately, the lack of PMR support in current compiler technology (both host and freestanding) is frustrating and is not currently a realistic option.

⁴⁶<https://godbolt.org/z/f4vn91zn>

Hopefully, we will see better support for string management in the future, but at the moment, unless you have support for C++17 stick to using NTBS. If you do have support for C++17, then it is probably best to still avoid `std::string` but look to utilise `std::string_view` where appropriate.

For dynamically constructed arrays, then until better support for PMR, `std::array` and `snprintf` are probably still the best options. However, the C++20 facility `std::format_to_n` looks to be a modern substitution to `snprintf`.

Chapter 63

Determining if a template specialization exists

• Lukas Barth  2023-01-01  ★★

One C++17 problem I come across every now and then is to determine whether a certain class or function template specialization exists - say, for example, whether `std::swap<SomeType>` or `std::hash<SomeType>` can be used. I like to have solutions for these kind of problems in a template-toolbox, usually just a header file living somewhere in my project. In this article I try to build solutions that are as general as possible to be part of such a toolbox.

Note that it is indeed not entirely clear what it means that “a specialization exists”. Even though this might seem unintuitive, I’ll postpone that discussion to the last section, and will for the start continue with the intuitive sense that “specialization exists” means “I can use it at this place in the code”.

Where I mention the standard, I refer to the C++17 standard, and I usually use GCC 12 and Clang 15 as compilers. See my note on MSVC at the end for why I’m not using it in this article.

Update, 3.1.2023: /u/gracicot commented on reddit about some limitations of the below solution. This revolves around testing for the same specialization multiple times, once before the specialization has been declared, once after. This is indeed not possible (and forbidden by the C++ standard). I have added a section with the respective warning. For now, let’s go with this handwavy rule: If you want to test in multiple places whether a template `Tmpl` is specialized for type `T`, the answer must be the same in all these places.

Update, 7.1.2023: An attentive reader spotted that at the end of the section with the generic solution for class templates, I write that I now drop the assumption that every class always has a default constructor - but then still rely on that default constructor. I forgot to put in a `std::declval` there. Thanks for the note!

Update, 15.2.2023: This article has been published in ACCU’s Overload issue 173.

63.1 Testing for a specific function template specialization

First, the easiest part: Testing for one specific function template specialization. I’ll use `std::swap` as an example here, though in C++17 you should of course use `std::is_swappable` to test for the existence

of `std::swap<T>`.

Without much ado, here's my proposed solution:

```

1 struct HasStdSwap {
2 private:
3     template <class T, class Dummy = decltype(std::swap<T>(std::declval<T &>(),
4                                         std::declval<T &>()))>
5     static constexpr bool exists(int) {
6         return true;
7     }
8
9     template <class T> static constexpr bool exists(char) { return false; }
10
11 public:
12     template <class T> static constexpr bool check() { return exists<T>(42); }
13 };

```

Let's unpack this: The two `exists` overloads are doing the heavy lifting here. The goal is to have the preferred overload when called with the argument `42` (i.e., the overload taking `int`) return `true` if and only if `std::swap<T>` is available. To achieve this, we must only make sure that this overload is not available if `std::swap<T>` does not exist, which we do by SFINAE-ing it away if the expression `decltype(std::swap<T>(std::declval<T&>(), std::declval<T&>()))` is malformed.

You can play with this here(<https://godbolt.org/z/8x689x1n1>) at Compiler Explorer. Note that we need to use `std::declval<T&>()` instead of the more intuitive `std::declval<T>()` because the result type of `std::declval<T>()` is `T&&`, and `std::swap` can of course not take rvalue references.

63.2 Testing for a specific class template specialization

Now that we have a solution to test for a specific function template specialization, let's transfer this to class templates. We'll use `std::hash` as an example here.

To transform the above solution, we only need to figure out what to use as default-argument type for `Dummy`, i.e., something that is well-formed exactly in the cases where we want the result to be `true`. We can't just use `Dummy = std::hash<T>`, because `std::hash<T>` is a properly declared type for all types `T`! What we actually want to check is whether `std::hash<T>` has been defined and not just declared. If a type has only been declared (and not defined), it is an incomplete type. Thus we should use something that does work for all complete types, but not for incomplete types.

In the case of `std::hash`, we can assume that every definition of `std::hash` must have a default constructor (as mandated by the standard for `std::hash`), so we can do this:

```

1 struct HasStdHash {
2 private:
3     template <class T, class Dummy = decltype(std::hash<T>{})>
4     static constexpr bool exists(int) {
5         return true;

```

```

6     }
7
8     template <class T>
9     static constexpr bool exists(char) {
10    return false;
11 }
12
13 public:
14     template <class T>
15     static constexpr bool check() {
16        return exists<T>(42);
17    }
18 };

```

This works nicely as you can see here(<https://godbolt.org/z/4TxPo76dT>) at Compiler Explorer. This is how you can use it:

```
std::cout << "Does std::string have std::hash? " << HasStdHash::check<std::string>();
```

63.3 A generic test for class templates

If I want to put this into my template toolbox, I can't have a implementation that's specific for `std::hash` (and one for `std::less`, one for `std::equal_to`, ...). Instead I want a more general form that works for all class templates, or at least those class templates that only take type template parameters.

To do this, I want to pass the class template to be tested as a template template parameter. Adapting our solution from above, this is what we would end up with:

```

1 template <template <class... InnerArgs> class Tmpl>
2 struct IsSpecialized {
3 private:
4     template <class... Args, class dummy = decltype(Tmpl<Args...>{})>
5     static constexpr bool exists(int) {
6         return true;
7     }
8     template <class... Args>
9     static constexpr bool exists(char) {
10        return false;
11    }
12
13 public:
14     template <class... Args>
15     static constexpr bool check() {
16        return exists<Args...>(42);
17    }
18 };

```

This does still work for `std::hash`, as you can see here(<https://godbolt.org/z/1PWq5ahWj>) at Compiler Explorer, when being used like this:

```
std::cout << "Does std::string have std::hash? "
    << IsSpecialized<std::hash>::check<std::string>();
```

However, by using `Tmpl<Args...>{}`, we assume that the class (i.e., the specialization we are interested in) has a default constructor, which may not be the case. We need something else that always works for any complete class, and never for an incomplete class.

If we want to stay with a type, we can use something unintuitive: the type of an explicit call of the destructor. While the destructor itself has no return type (as it does not return anything), the standard states in [expr.call]:

If the postfix-expression designates a destructor, the type of the function call expression is void;
[...]

So this will work regardless of how the template class is defined¹(changes highlighted):

```
1  template <template <class... InnerArgs> class Tmpl>
2  struct IsSpecialized {
3  private:
4      template <class... Args,
5                  class dummy = decltype(std::declval<Tmpl<Args...>() . ~Tmpl<Args...>()))
6      static constexpr bool exists(int) {
7          return true;
8      }
9      template <class... Args>
10     static constexpr bool exists(char) {
11         return false;
12     }
13
14 public:
15     template <class... Args>
16     static constexpr bool check() {
17         return exists<Args...>(42);
18     }
19 };
```

Note that we use `std::declval` to get a reference to `Tmpl<Args...>` without having to rely on its default constructor. Again you can see this at work at Compiler Explorer(<https://godbolt.org/z/vYaoMYTWq>).

¹With the notable exception of the template class having a private destructor.

63.4 Problem: Specializations that sometimes exist and sometimes don't

The question of whether `SomeTemplate<SomeType>` is a complete type (a.k.a. “the specialization exists”) depends on whether the respective definition has been seen or not. Thus, it can differ between translation units, but also within the same translation unit. Consider this case:

```

1 template<class T> struct SomeStruct;
2
3 bool test1 = IsSpecialized<SomeStruct>::check<std::string>();
4
5 template<> struct SomeStruct<std::string> {};
6
7 bool test2 = IsSpecialized<SomeStruct>::check<std::string>();

```

What should happen here? What values would we want for `test1` and `test2`? Intuitively, we would want `test1` to be `false`, and `test2` to be `true`. If we try to square this with the `IsSpecialized` template from above, something weird happens: The same template, `IsSpecialized<SomeStruct>::check<std::string>()`, is instantiated with the same template arguments but should emit a different behavior. Something cannot be right here. If you imagine both tests (once with the desired result `true`, once with desired result `false`) to be spread across different translation units, this has the strong smell of an ODR-violation.

If we try this at Compiler Explorer(<https://godbolt.org/z/Y1zYn9Tqj>), we indeed see that this does not work. So, what's going on here?

The program is actually ill-formed, and there's nothing we can do to change that. The standard states in `temp.expl.spec/6`:

If a template [...] is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required. [...]

Of course the test for the availability of the specialization would “cause an implicit instantiation” (which fails and causes SFINAE to kick in).² Thus it is *always* ill-formed to have two tests for the presence of a specialization if one of them “should” succeed and one “should” fail.

In fact, the standard contains a paragraph, `temp.expl.spec/7`, that does not define anything (at least if I read it correctly), but only issues a warning that ‘there be dragons’ if one has explicit specializations sometimes visible, sometimes invisible. I’ve not known the standard to be especially poetic, this seems to be the exception:

The placement of explicit specialization declarations [...] can affect whether a program is well-formed according to the relative positioning of the explicit specialization declarations and their points of instantiation in the translation unit as specified above and below. When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation.

²This is explicitly stated in [temp.inst]/6

Thus, as a rule of thumb (not just for testing whether a specialization exists): If you use `Tmpl<T>` at multiple places in your program, you must make sure that any explicit specialization for `Tmpl<T>` is visible at **all** those places.

63.5 A generic test for function templates

The move from testing whether one *particular* class template was specialized for a type `T` to having a test for *arbitrary* class templates was pretty easy. Unfortunately it is a lot harder to replicate the same for function templates. This is mainly because we cannot pass around function templates as we can pass class templates as template template parameters.

If we want to have a template similar to `IsSpecialized` from above (let's call it `FunctionSpecExists`), we need a way of encapsulating a function template so that we can pass it to our new `FunctionSpecExists`. On the other hand, we want to keep this "wrapper" as small as possible, because we will need it at every call site. Thus, building a struct or class is not the way to go.

C++14 generic lambdas provide a neat way of encapsulating a function template. Remember that a lambda expression is of (an unnamed) class type. Thus, we can pass them around as template parameter, like any other type.

Encapsulating the function template we are interested in (`std::swap`, again) in a generic lambda could look like this:

```
auto l = [](auto &lhs, auto &rhs) { return std::swap(lhs, rhs); };
```

Now that we have something that is callable if and only if `std::swap<decltype(lhs)>` is available. When I write "is callable if", this directly hints at what we can use to implement our `FunctionSpecExists` struct - "is callable" sounds a lot like `std::is_invocable`, right?

So, to test whether `SomeType` can be swapped via `std::swap`, can we just do this?

```
auto l = [](auto &lhs, auto &rhs) { return std::swap(lhs, rhs); };
bool has_swap = std::is_invocable_v<decltype(l)>(l, SomeType &, SomeType &);
```

Unfortunately, no(<https://godbolt.org/z/hjs0Wz7G9>). Assuming that `SomeType` is not swappable, we are getting no matching call to `std::swap` errors. The problem here is that `std::is_invocable` must rely on SFINAE to remove the infeasible `std::swap` implementations (which in this case are all implementations). However, SFINAE only works in the elusive "immediate context" as per [temp.deduct]/8. The unnamed class that the compiler internally creates for the generic lambda looks (simplified) something like this:

```
1 struct Unnamed {
2     template <class T1, class T2>
3     auto operator()(T1 &lhs, T2 &rhs) {
4         return std::swap(lhs, rhs);
5     }
6 };
```

Here it becomes obvious that plugging in `SomeType` for `T1` and `T2` does not lead to a deduction failure in the "immediate context" of the function, but actually just makes the body of the `operator()` func-

tion ill-formed. We need the problem (no matching `std::swap`) to kick in in one of the places for which [temp.deduct] says that types are substituted during template deduction. Quoting from [temp.deduct]/7:

The substitution occurs in all types and expressions that are used in the function type and in template parameter declarations.

One thing that is part of the function type is a *trailing return type*, so we can use that. Let's rewrite our lambda to:

```
1 auto betterL = [](auto &lhs, auto &rhs) -> decltype(std::swap(lhs, rhs)) {
2     return std::swap(lhs, rhs);
3 };
```

Now we have a case where, if you were to substitute the non-swappable `SomeType` for the `auto` types, there is an error in the types involved in the function type. And indeed, this actually works, as you can see here(<https://godbolt.org/z/5z7x1nq6G>) on Compiler Explorer:

```
1 auto betterL = [](auto &lhs, auto &rhs) -> decltype(std::swap(lhs, rhs)) {
2     return std::swap(lhs, rhs);
3 };
```

4 **constexpr bool** sometype_has_swap =
5 `std::is_invocable_v<decltype(betterL), SomeType &, SomeType &>;`

I don't think that you can further encapsulate this into some utility templates to make the calls more compact, so that's just what I will use from now on.

63.6 What do I mean by “a specialization exists”

I wrote at the beginning that it's not entirely clear what “a specialization exists” should even mean. It is of course not possible - neither for class templates, nor for function templates - to check at compile time whether a certain specialization exists somewhere, which may be in a different translation unit. I wrote the previous sections with the aim of testing whether the class template (resp. function template) can be “used” with the given arguments at the point where the test happens.

For **class templates**, I say a “specialization exists” if, for a given set of template arguments, the resulting type is not just declared, but also defined (i.e., it is a complete type). As an example:

```
1 template<class T>
2 struct SomeStruct;
3
4 template<>
5 struct SomeStruct<int> {};
6
7 // (Point A) Which specializations "exist" at this point?
8
9 template<>
10 struct SomeStruct<std::string> {};
```

In this code, at the marked line, only the specialization for the type `int` “exists” .

For **function templates**, it’s actually a bit more complicated, since C++ has no concept of “incomplete functions” analogous to “incomplete types” . Here, I say that a specialization “exists” if the respective overload has been declared. Take this example:

```

1 template<class T>
2 void doFoo(T t);
3
4 template<class T, class Dummy=std::enable_if_t<std::is_integral_v<T>, bool> = true>
5 void doBar(T t);
6 template<class T, class Dummy=std::is_same_v<T, std::string>, bool> = true>
7 void doBar(T t) {};
8
9 // (Point B) Which specializations "exist" at this point?

```

At the marked, line:

- For any type `T`, the specialization `doFoo<T>` “exists” , because the respective overload has been declared in lines one and two.
- The two specializations `doBar<std::string>` and `doBar<T>` for any integral type `T` “exist” . Note that this is independent of whether the function has been defined (like `doBar<std::string>`) or merely declared.
- For all non-integral, non-`std::string` types `T`, the specialization `doBar<T>` does “not exist” .

This of course means that our “test for an existing specialization” for functions is more of a “test for an existing overload” , and can in fact be used to achieve this.

63.7 A note on MSVC and std::hash

In all my examples, I used GCC and Clang as compilers. This is because my examples for `std::hash` do not work with MSVC(<https://godbolt.org/z/Mj8j7zErM>), at least if you enable C++17 (it works in C++14 mode). That is because of this (simplified) `std::hash` implementation in MSVC’s STL implementation:

```

1 template <class _Kty, bool _Enabled>
2 struct _Conditionally_enabled_hash { // conditionally enabled hash base
3     size_t operator()(&_Kty &_Keyval) const
4     {
5         return hash<_Kty>::_Do_hash(_Keyval);
6     }
7 };
8
9 template <class _Kty>
10 struct _Conditionally_enabled_hash<_Kty,
11                               false> { // conditionally disabled hash base
12     _Conditionally_enabled_hash() = delete;

```

```
13     // *no* operator()!
14 };
15
16 template <class _Kty>
17 struct hash
18     : _Conditionally_enabled_hash<_Kty, should_be_enabled_v<_Kty>>
19 {
20     // *no* operator()!
21 };
```

This implementation is supposed to handle all integral, enumeration and pointer types (which is what `should_be_enabled_v` tests for), but the point is: For all other types, this gives you a defined, and thus complete, class - which does not have an `operator()`. I'm not sure why the designers built this this way, but that means that on MSVC, our testing-for-type-completeness does not work to determine whether a type has `std::hash`. You must also test whether `operator()` exists!

Chapter 64

How to Store an lvalue or an rvalue in the Same Object

👤 Jonathan Boccaro 📅 2022-05-16 💬 ★★

There seems to be a problem coming up every so often C++ code: how can an object keep track of a value, given that this value can come from either an lvalue or an rvalue?

In short, if we keep the value as a reference then we can't bind to temporary objects. And if we keep it as a value, we incur unnecessary copies when it is initialized from an lvalue.

What's a C++ programmer to do?

There are several ways to cope with this situation. I find that using `std::variant` offers a good trade-off to have expressive code.

64.1 Keeping track of a value

Here is a more detailed explanation of the problem.

Consider a class `MyClass`. We would like to give `MyClass` access to a certain `std::string`. How do we represent the string inside of `MyClass`?

We have two options:

- storing it as a reference,
- storing it as a value.

64.1.1 Storing a reference

If we store it as a reference, for example a `const` reference:

```
1 class MyClass
2 {
3     public:
4         explicit MyClass(std::string const& s) : s_(s) {}
5         void print() const
```

```

6     {
7         std::cout << s_ << '\n';
8     }
9 private:
10    std::string const& s_;
11 };

```

Then we can initialize our reference with an lvalue:

```

std::string s = "hello";
MyClass myObject{s};
myObject.print();

```

This code prints out:

```
hello
```

All good. But what if we want to initialize our object with an rvalue? For example with this code:

```

MyClass myObject{std::string{"hello"}};
myObject.print();

```

Or with this code:

```

std::string getString(); // function declaration returning by value

MyClass myObject{getString()};
myObject.print();

```

Then the code has **undefined behaviour**. Indeed, the temporary string object is destroyed on the same statement it is created. When we call print, the string has already been destroyed and using it is illegal and leads to undefined behaviour.

64.1.1.1 Really?

To illustrate this, if we replace `std::string` with a type `X` where we log in the destructor:

```

1 struct X
2 {
3     ~X() { std::cout << "X destroyed" << '\n'; }
4 };
5
6 class MyClass
7 {
8 public:
9     explicit MyClass(X const& x) : x_(x) {}
10    void print() const
11    {
12        // using x_;

```

```

13     }
14 private:
15     X const& x_;
16 };

```

Let's also add logging to the call site:

```

 MyClass myObject(X{});
 std::cout << "before print" << '\n';
 myObject.print();

```

This program then prints (live code here <https://godbolt.org/z/ejaMze>):

```

X destroyed
before print

```

We can see that the object is destroyed before we attempt to use it.

64.1.2 Storing a value

The other option we have is to store a value. This allows us to use move semantics to move the incoming temporary into the stored value:

```

1 class MyClass
2 {
3 public:
4     explicit MyClass(std::string s) : s_(std::move(s)) {}
5     void print() const
6     {
7         std::cout << s_ << '\n';
8     }
9 private:
10    std::string s_;
11 };

```

Now with this call site:

```

MyClass myObject{std::string{"hello"}};
myObject.print();

```

We incur two moves (one to construct `s` and one to construct `s_`) and we don't have undefined behaviour. Indeed, even if the temporary is destroyed, `print` uses the instance inside of the class.

Unfortunately, if we go back to our first call site, with an lvalue:

```

std::string s = "hello";
MyClass myObject{s};
myObject.print();

```

Then we’re no longer making two moves: we’re making one copy (to construct `s`) and one move (to construct `s_`).

What’s more, our purpose was to give `MyClass` access to the string, and if we make a copy we have a different instance than the one that came in. So they won’t be in sync.

With the temporary object it wasn’t a problem because it was to be destroyed anyway and we moved it in just before, so we still had access to “that” string. But by making a copy we no longer give `MyClass` access to the incoming string.

So using a value is not a good solution either.

64.2 Storing a variant

Storing a reference is not a good solution, and storing a value is not a good solution either. What we would like to do is to store a reference if the value is initialised from an lvalue, and store a value if it is stored from an rvalue.

But a data member can only be of one type: value or reference, right?

Well, with a `std::variant`, it can be either one.

However, if we try to store a reference in a variant, like this:

```
std::variant<std::string, std::string const&>
```

We get an compilation error expressed with a broken static assert:

```
variant must have no reference alternative
```

To achieve our purpose we need to put our reference inside of another type.

This means that we have to write specific code to handle our data member. If we write such code for `std::string` we won’t be able to use it for another type.

At this point it would be good to write the code in a generic way.

64.3 A generic storage class

The storage of our motivating case needed to be either a value or a reference. Since we’re writing this code for a general purpose now, we may as well allow non-const references too.

Since the variant cannot hold references directly, let’s store them into wrappers:

```
1 template<typename T>
2 struct NonConstReference
3 {
4     T& value_;
5     explicit NonConstReference(T& value) : value_(value){};
6 };
7
8 template<typename T>
9 struct ConstReference
10 {
```

```

11     T const& value_;
12     explicit ConstReference(T const& value) : value_(value){};
13 }
14
15 template<typename T>
16 struct Value
17 {
18     T value_;
19     explicit Value(T&& value) : value_(std::move(value)) {}
20 };

```

And let's define our storage to be either one of those cases:

```

template<typename T>
using Storage = std::variant<Value<T>, ConstReference<T>, NonConstReference<T>>;

```

Now we need to give access to the underlying value of our variant, by providing a reference. We create two types of access: one const and one not const.

64.3.1 Defining const access

To define const access, we need to make each of the three possible type inside of the variant produce a const reference.

To access data inside the variant, we'll use `std::visit` and the canonical *overload pattern*, which can be implemented in C++17 the following way:

```

1 template<typename... Functions>
2 struct overload : Functions...
3 {
4     using Functions::operator()...;
5     overload(Functions... functions) : Functions(functions)... {}
6 };

```

To get our const reference, we can just create one for each case of the variant:

```

1 template<typename T>
2 T const& getConstReference(Storage<T> const& storage)
3 {
4     return std::visit(
5         overload(
6             [] (Value<T> const& value) -> T const& { return value.value_; },
7             [] (NonConstReference<T> const& value) -> T const& { return value.value_; },
8             [] (ConstReference<T> const& value) -> T const& { return value.value_; }
9         ),
10        storage
11    );
12 }

```

64.3.2 Defining non-const access

The creation of a non const reference uses the same technique, except that if is variant is a ConstReference, it can't produce a non-const reference. However, when we std::visit a variant, we have to write code for each of its possible types:

```

1 template<typename T>
2 T& getReference(Storage<T>& storage)
3 {
4     return std::visit(
5         overload(
6             [] (Value<T>& value) -> T& { return value.value_; },
7             [] (NonConstReference<T>& value) -> T& { return value.value_; },
8             [] (ConstReference<T>& ) -> T& { /* code handling the error! */ }
9         ),
10        storage
11    );
12 }
```

We should never end up in that situation, but we still have to write some code for it. The first idea that comes to (my) mind is to throw an exception:

```

1 struct NonConstReferenceFromReference : public std::runtime_error
2 {
3     explicit NonConstReferenceFromReference(std::string const& what)
4         : std::runtime_error{what} {}
5 };
6
7 template<typename T>
8 T& getReference(Storage<T>& storage)
9 {
10     return std::visit(
11         overload(
12             [] (Value<T>& value) -> T& { return value.value_; },
13             [] (NonConstReference<T>& value) -> T& { return value.value_; },
14             [] (ConstReference<T>& ) -> T& {
15                 throw NonConstReferenceFromReference{
16                     "Cannot get a non const reference from a const reference" } ;
17             }
18         ),
19        storage
20    );
21 }
```

If you have other suggestions, I'd love to hear them!

64.4 Creating the storage

Now that we have defined our storage class, let's use it in our motivating case to give access to the incoming `std::string` regardless of its value category:

```

1  class MyClass
2  {
3      public:
4          explicit MyClass(std::string& value) : storage_(NonConstReference(value)){}
5          explicit MyClass(std::string const& value) : storage_(ConstReference(value)){}
6          explicit MyClass(std::string&& value) : storage_(Value(std::move(value))){}
7
8      void print() const
9      {
10         std::cout << getConstReference(storage_) << '\n';
11     }
12
13  private:
14      Storage<std::string> storage_;
15  };

```

Consider the first call site, with an lvalue:

```

std::string s = "hello";
MyClass myObject{s};
myObject.print();

```

It matches the first constructor, and creates a `NonConstReference` inside of the `storage` member. The non-const reference is converted into a const reference when the `print` function calls `getConstReference`.

Now consider the second call site, with the temporary value:

```

MyClass myObject{std::string{"hello"}};
myObject.print();

```

This one matches the third constructor, and moves the value inside of the `storage`. `getConstReference` then returns a const reference to that value to the `print` function.

64.5 The evolution of the standard library

`std::variant` offers a very adapted solution to the classical problem of keeping track of either an lvalue or an rvalue in C++.

The code of this technique is expressive because `std::variant` allows to express something that is very close to our intention: “depending on the context, the object could be either this or that”. In our case, “this” and “that” are a “reference” or a “value”.

Before C++17 and `std::variant`, solving that problem was tricky and led to code that was difficult to write correctly. With the language evolving, the standard library gets more powerful and lets us express our intentions with more and more expressive code.

We will see other ways in which the evolution of the standard library helps us write more expressive code in a future article. Stay tuned!

Chapter 65

Automatic Serialization in C++ for Game Engines

• Deckhead 📅 2022-03-28 🔮 ★★

You've made your game, probably made an engine, and now you're ready to put on the finishing touches. Players are definitely going to want to save their game. So, that means storing all the current game state to a file... Wow, that's a lot of work, right? Now every enemy, character, item, fallen tree, exploded building, location etc all needs to be put into a file. And then you need to read it back again. What a nightmare.

In this article, I'm going to show you how I setup my very own serialisation library. Sure, there's existing libraries you can use, but like always, I don't even look at them, I'm more interested in doing it myself.

65.1 Caveats

Of course, there's a few things we're going to have to align on, right off the bat, for this to work. The following list is a few concessions we need to make in order to allow for easy serialisation. I'll justify each as well.

65.1.1 References and Pointers can't be serialised

Oh, that's a big one. Let's get it right out the way. You can serialise references and pointers, I've done it before, but it's a nightmare. For example, let's say we have some objects like so (don't worry about the poor design choices for now):

```
1 class Player
2 {
3     public:
4         int m_health;
5     };
6
```

```

7  class Enemy
8  {
9  public:
10    Player* m_targetPlayer;
11    int m_health;
12 };
13
14 int main()
15 {
16    Player[2] players;
17    Enemy[5] enemies;
18
19    enemies[0].m_targetPlayer = &players[0];
20
21    ...
22
23 }

```

Now, if you hot-save in the middle of the game, how do we store the fact that **Enemy 1** is currently targeting **Player 1**? Our game data is using a pointer to the targeted player. We can't just write that pointer to the file and reload it later, because the memory locations when we reload the game are all different, so that won't work. We can't write a copy of **Player 1** to the file, because we don't want to reload and create a second instance of player 1.

That's the difficulty. But you can work around it the hard way.

When **serialising** an object, if you come across a pointer you could:

- Add it to a list of “pointers yet to be serialised”
- Make a reference in the serialisation of the current object that it needs the new address for this pointed to object

After you've finished serialising, you should have all objects serialised, and anything that was referencing an pointer/reference should have serialised the original address. When you serialise objects, include the current address of it. In the end, you might have a few “dangling” objects that only ever existed on the heap, in which case you can serialize them last.

When **deserializing**, anything that says “oh my pointer referenced this original memory address”, you can find the already deserialized object with that original memory address; or delay for when you do have it but keeping long pointers to yet-to-be-given-the-correct-address pointers. When something is deserialized, you can go to all those long-pointers and point them to this new memory address.

Essentially it's a two-pass method. Pass one, read objects you can and leave reminders for pointers. Pass two, patch the pointers.

65.1.1.1 Alternative

All in all, it's messy, hard to debug, and error prone to utilize pointers and serialize. In respect to engine speed and extensibility, you really shouldn't be throwing pointers around anyway. It can be done

but it's horrible. And if you're using an Entity Component System¹, you very likely have no pointers to worry about.

Note

An extra note, if you have polymorphic pointers, you need to also keep track of the true class when serializing (override virtual functions). Deserialization will require a map in your code and a special factory function to create the correct sub-class. More and more pain.

Aside from ECS, you can also do away with pointers by using some other method of indirection. Identifiers that map to a unique instance of an object is often good, with a globally accessible map of identifiers to actual pointers. That can be deserialized separately to everything else, and if all your objects use identifiers instead of actual pointers, everything will go together again. Of course, speed issues exist here when looking up the pointer, but you should be reducing pointer use anyway.

In a future article, I will cover the cases of serialisation of pointers.

65.1.2 Objects need to be default constructible and copyable

This is a bit of a burden, but the way I have things laid out here, anything that's serialised needs to be default constructible (or otherwise you can construct a "dummy" one), and copyable.

For me and my purposes, this isn't a big problem. The primary reason is my reliance on an Entity Component System, which makes zeroing out my structs super easy and non-problematic.

But maybe things aren't as simple for you. You can work around these limitations by adjusting the code that does the serialisation (you'll see it later). It shouldn't be too burdensome to do something with a "Loader" or "Loading Function" that investigates the file for the next "class" to be constructed and handle things from there; potentially even passing the stream as a constructor argument for the class itself.

65.1.3 You have to add some special code to your classes

Caveat 2 isn't as bad. If you want a class to be "serializable", you're going to have to add some special code to it. I've seen this done a few different ways, for example, inheriting from a "Serializable" class. The way we're going to do it is hopefully a little less annoying.

The reason you must always add some special code is that C++, as of C++20 anyway, offers no real reflection capability. What reflection allows you to do is give the program code dynamic knowledge of what is in the code. Like how in JavaScript everything is a string, so you can parse a function and edit it at runtime so the function does something else.

We haven't got that. Without it, any serialisation code we write must be told, by you, what the object looks like (what members it has to serialise). I've seen a few different ways that people add this:

1. Code/Scripts that reads the source code headers of your project, and generates serialisation automatically. This is really cool, but personally I find it sort too...recursive...strange...idk. It sits wrong with me.
2. Inheriting from a "Serializable" class that requires you to specialise a "Serialise" and "Deserialize" method. Not great, you're basically just writing all the code yourself. It's a naïve approach, but

¹<https://indiegamedev.net/2020/05/19/an-entity-component-system-with-data-locality-in-cpp/>

works very well for small projects.

We’re going to use MACROS. Actually, let’s get started.

65.2 Why use Preprocessor Macros for C++ Serialization?

The aim of this article is to use the pre-processor to write our serialisation code for us. Yes, that’s right, we’ll write a program to write our program.

First of all; here’s the naïve approach mentioned above (sans inheritance from a base `Serializable` class, we don’t really need to do that):

```

1  class ObjectA
2  {
3      public:
4          std::string m_str;
5          int m_int;
6          float m_float;
7
8      void Serialise(std::ostream& stream) const
9      {
10         stream << m_str << " " << m_int << " " << m_float << " ";
11     }
12
13     void Deserialise(std::istream& stream)
14     {
15         stream >> m_str >> m_int >> m_float;
16     }
17 };

```

We create an `std::ofstream` and pass it to the `Serialise` method and we’ll have an output. That same output back through an `std::ifstream` and passed to `Deserialise` will recreate the object.

Like I said, for simple use-cases, this works well. We might even change it up a bit to be fancier:

```

1  class ObjectA
2  {
3      public:
4          std::string m_str;
5          int m_int;
6          float m_float;
7
8      friend std::ostream& operator<<(std::ostream& stream, const ObjectA& obj)
9      {
10         stream << obj.m_str << " " << obj.m_int << " " << obj.m_float << " ";
11         return stream;
12     }

```

```

13
14     friend std::istream& operator>>(std::istream& stream, ObjectA& obj)
15     {
16         stream >> obj.m_str >> obj.m_int >> obj.m_float;
17         return stream;
18     }
19 };

```

And in this way we can invoke it directly on a stream, `outputFile objectAInstance`; and it works well.

The big issue is that we have to do this for every kind of object. When you have a tonne of different things that need to be serialised... well that's a big ask. Our preprocessor macros are going to write the above code for us, that's why we use it.

Tips

Be wary of differences in compilers with preprocessor macros. They're supposed to be somewhat standard but they aren't always. In particular, I know that various version of MSVC++ compiler can be troublesome.

65.3 Building Blocks

First, let's create a couple of objects we want to be able to serialise. `Thing1` and `Thing2`.

```

1  class Thing1
2  {
3      public:
4
5          Thing1()
6          :
7              m_int(0),
8              m_float(0),
9              m_string("something"),
10             m_vecs(4)
11         {
12
13     }
14
15     private:
16         int m_int;
17         float m_float;
18         std::string m_string;
19         Something2 m_something;
20         std::vector<Something2> m_vecs;
21     };

```

```

1  class Thing2
2  {
3  public:
4      Thing2()
5      :
6          m_int(0),
7          m_int2(0),
8          m_vec({3.2f,.4f,1224.3f})
9      {
10
11  }
12
13  int m_int;
14  int m_int2;
15
16  std::vector<float> m_vec;
17 };

```

These are a little silly, but they'll illustrate the point well. We're going to want our serialization to serialize `m_int`, `m_float`, `m_string`, and `m_thing2` from `Thing1`; and we want it to serialize `m_int` and `m_int2` from `Thing2`.

When our macros insert code, we need to provide it with the code to ultimately insert. If we want to be able to serialise different types, we need to provide functions to do so. Let's start with:

```

1  template<typename StreamType, typename T>
2  requires std::derived_from<StreamType, std::ostream>
3  void serialise(StreamType& s, const T& t)
4  {
5      s << t << " ";
6 }

```

Easy and simple enough I think. Serialisation requires some kind of output stream, and simply puts it out.

Note

Note that I'm adding a space. That's because I'm going to serialise to text. You might want to serialise to binary. Heck, you might want to serialise to JSON or YAML or XML. So think about that when you adopt this code for your own game.

I've chosen simple text output because it's easy and I can manipulate values before reloading. However, JSON or some other structured text format would be easier to make sense of. Binary would probably save space.

To reverse the above serialisation we would:

```

1  template<typename StreamType, typename T>
2  requires std::derived_from<StreamType, std::istream>

```

```

3 void deserialise(StreamType& s, T& t)
4 {
5     s >> t;
6 }
```

No issues, very easy.

Oh but wait. We need to be able to serialize `std::vector` and `std::string` as well, plus reload them. We'll need to add some specialised versions of these functions:

```

1 template<typename StreamType, typename T, typename Alloc>
2 requires
3     std::derived_from<StreamType, std::ostream> &&
4     std::same_as<std::basic_string<char, std::char_traits<char>, Alloc>, T>
5 void serialise(StreamType& s, const T& t)
6 {
7     s << t.size() << " " << t;
8 }
9
10 template<typename StreamType, typename T, typename Alloc>
11 requires std::derived_from<StreamType, std::istream> &&
12 std::same_as<std::basic_string<char, std::char_traits<char>, Alloc>, T>
13 void deserialise(StreamType& s, T& t)
14 {
15     t = "";
16
17     typename T::size_type len;
18     s >> len;
19
20     for(std::size_t i = 0; i < len; ++i)
21     {
22         typename T::value_type c;
23         s >> c;
24         t += c;
25     }
26 }
27
28
29
30 template<typename StreamType, typename T, typename Alloc>
31 requires
32     std::derived_from<StreamType, std::ostream>
33 void serialise(StreamType& s, const std::vector<T, Alloc>& t)
34 {
35     s << t.size() << " ";
```

```

36     for(const T& tt : t)
37     {
38         s << tt << " ";
39     }
40 }
41
42 template<typename StreamType, typename T, typename Alloc>
43 requires
44     std::derived_from<StreamType, std::istream>
45 void deserialise(StreamType& s, std::vector<T, Alloc>& t)
46 {
47     using VecType = std::vector<T, Alloc>;
48     t.clear();
49
50     typename VecType::size_type len;
51     s >> len;
52     for(std::size_t i = 0; i < len; ++i)
53     {
54         typename VecType::value_type c;
55         s >> c;
56         t.push_back(c);
57     }
58 }
```

To serialize a string, you need to first output how many characters it has. That way, when you deserialize the same string, you know how many characters to read back in. An `std::vector` is much the same, serialize the number of elements so you know how many to read back in.

Tips

You might be wondering why I've got those `Alloc` arguments in there. Well, if you're using custom allocators with the STL containers, you'll need them like the above to be compatible. And if you're not using custom allocators, like how our `Thing1` and `Thing2` classes aren't, then the calls still work.

I won't write serialization functions for all the STL Containers, but that's how it works. You can easily write your own functions for the other containers you're using.

65.4 Looping Preprocessor Macro

It'll be clearer why we need it later, but we need some preprocessor code that will allow us to loop through variadic preprocessor arguments and output some constant arguments.

I.e., I could write `MACRO_COMMAND(1, 2, 3, 4)` or `MACRO_COMMAND(1, 2, 3, 4, 5, 6)` or `MACRO_COMMAND(1)`. I need to be able to send a variable number. The following achieves this. I'

I'll explain it now, but it probably won't click in your head for a little bit.

```

1 #define CONCATENATE(arg1, arg2)    CONCATENATE1(arg1, arg2)
2 #define CONCATENATE1(arg1, arg2)   CONCATENATE2(arg1, arg2)
3 #define CONCATENATE2(arg1, arg2)   arg1##arg2
4
5 #define FOR_EACH_1(what, o, i, x)      \
6     what(o, i, x)
7
8 #define FOR_EACH_2(what, o, i, x, ...)  \
9     what(o, i, x);                   \
10    FOR_EACH_1(what, o, i, __VA_ARGS__)
11
12 #define FOR_EACH_3(what, o, i, x, ...)  \
13     what(o, i, x);                   \
14    FOR_EACH_2(what, o, i, __VA_ARGS__)
15
16 #define FOR_EACH_4(what, o, i, x, ...)  \
17     what(o, i, x);                   \
18    FOR_EACH_3(what, o, i, __VA_ARGS__)
19
20 #define FOR_EACH_5(what, o, i, x, ...)  \
21     what(o, i, x);                   \
22    FOR_EACH_4(what, o, i, __VA_ARGS__)
23
24 #define FOR_EACH_6(what, x, ...)        \
25     what(o, i, x);                   \
26    FOR_EACH_5(what, o, i, __VA_ARGS__)
27
28 #define FOR_EACH_7(what, o, i, x, ...)  \
29     what(o, i, x);                   \
30    FOR_EACH_6(what, o, i, __VA_ARGS__)
31
32 #define FOR_EACH_8(what, o, i, x, ...)  \
33     what(o, i, x);                   \
34    FOR_EACH_7(what, o, i, __VA_ARGS__)
35
36 #define FOR_EACH_NARG(...) FOR_EACH_NARG__(__VA_ARGS__, FOR_EACH_RSEQ_N())
37 #define FOR_EACH_NARG__(...) FOR_EACH_ARG_N(__VA_ARGS__)
38 #define FOR_EACH_ARG_N(_1, _2, _3, _4, _5, _6, _7, _8, N, ...) N
39 #define FOR_EACH_RSEQ_N() 8, 7, 6, 5, 4, 3, 2, 1, 0
40
41 #define FOR_EACH_(N, what, ...) CONCATENATE(FOR_EACH_, N)(what, __VA_ARGS__)

```

```
42 #define FOR_EACH(what, o, i, ...) \
43 FOR_EACH_(FOR_EACH_NARG(__VA_ARGS__), what, o, i, __VA_ARGS__)
```

What this let's me do is make a call like:

```
FOR_EACH(MACRO_COMMAND, repeated_arg_o, repeated_arg_i,
          for_each_arg1, for_each_arg2, for_each_arg3)
```

and the preprocessor would replace it with:

```
MACRO_COMMAND(repeated_arg_o, repeated_arg_i, for_each_arg1)
MACRO_COMMAND(repeated_arg_o, repeated_arg_i, for_each_arg2)
MACRO_COMMAND(repeated_arg_o, repeated_arg_i, for_each_arg3)
```

Of course, then it would replace `MACRO_COMMAND` with the actual thing, but you can see how it will loop through a variable number of arguments, keeping the first two consistent.

65.5 Operator » and operator « Macro Replacements

Now you can see where this is going. We need to add a `friend std::ostream& operator<<()` and `friend std::istream& operator>>()` method to all classes we wish to serialise. That way we can write:

```
1 Thing1 thing1;
2 std::ofstream file;
3 ...
4
5 file << thing1;
```

and it will all work. The magic in this is we want to take our class definitions from above, and add a single line of code:

```
1 class Thing2
2 {
3 public:
4     Thing2()
5     :
6         m_int(0),
7         m_int2(0),
8         m_vec({1.3f, 1.4f, 2.5f})
9     {
10 }
11
12     int m_int;
13     int m_int2;
14     std::vector<float> m_vec;
```

```

16
17     SERIALISE(Something2, m_int, m_int2, m_vec)
18 };
19
20 class Thing1
21 {
22 public:
23
24     Thing1()
25     :
26         m_int(0),
27         m_float(0),
28         m_string("something"),
29         m_vecs(4)
30     {
31
32 }
33
34     SERIALISE(Something, m_int, m_float, m_string, m_something, m_vecs)
35
36 private:
37     int m_int;
38     float m_float;
39     std::string m_string;
40     Something2 m_something;
41     std::vector<Something2> m_vecs;
42 };

```

So we need some MACROs that will take those two lines, and replace it with serialisation friend functions.

```

1 #define MEMBER_SERIALISE(outputStream, instance, memberName) \
2     serialise(outputStream, instance.memberName);
3
4 #define MEMBER_DESERIALISE(inputStream, instance, memberName) \
5     deserialise(inputStream, instance.memberName);
6
7
8 #define SERIALISE(className, ...) \
9     friend std::ostream& operator<<(std::ostream& outputStream, \
10                                         const className& instance) \
11 {
12     FOR_EACH(MEMBER_SERIALISE, outputStream, instance, __VA_ARGS__) \
13     return outputStream;

```

```

14 }
15
16 friend std::istream& operator>>(std::istream& inputStream,
17                                     className& instance)
18 {
19     FOR_EACH(MEMBER_DESERIALISE, inputStream, instance, __VA_ARGS__)
20     return inputStream;
21 }
```

And that's it! When you call **SERIALISE** you provide the class name, followed by a list of member variables that should be included in the serialisation. Because C++ has no reflection capability, you have to do this manual “this is what I want serialised”, but it’s not a big ask (just a single line of code).

If you take the above two classes and execute `g++ -E myfile.cpp` it will output the code after all the pre-processor commands have taken place. The relevant code, formatted nicer for humans (new lines characters and tabs), gives us the following:

```

1 class Something2
2 {
3 public:
4     Something2()
5     :
6         m_int(0),
7         m_int2(0),
8         m_vec({1.3f, 1.4f, 2.5f})
9     {
10
11 }
12
13 int m_int;
14 int m_int2;
15 std::vector<float> m_vec;
16
17 friend std::ostream& operator<<(std::ostream& outputStream, const Something2& instance)
18 {
19     serialise(outputStream, instance.m_int);
20     serialise(outputStream, instance.m_int2);
21     serialise(outputStream, instance.m_vec);
22     return outputStream;
23 }
24
25 friend std::istream& operator>>(std::istream& inputStream, Something2& instance)
26 {
27     deserialise(inputStream, instance.m_int);
28     deserialise(inputStream, instance.m_int2);
```

```
29         deserialise(inputStream, instance.m_vec);
30         return inputStream;
31     }
32 }
33
34 class Something
35 {
36 public:
37
38     Something()
39     :
40         m_int(0),
41         m_float(0),
42         m_string("something"),
43         m_vecs(4)
44     {
45
46 }
47
48 friend std::ostream& operator<<(std::ostream& outputStream, const Something& instance)
49 {
50     serialise(outputStream, instance.m_int);
51     serialise(outputStream, instance.m_float);
52     serialise(outputStream, instance.m_string);
53     serialise(outputStream, instance.m_something);
54     serialise(outputStream, instance.m_vecs);
55     return outputStream;
56 }
57
58 friend std::istream& operator>>(std::istream& inputStream, Something& instance)
59 {
60     deserialise(inputStream, instance.m_int);
61     deserialise(inputStream, instance.m_float);
62     deserialise(inputStream, instance.m_string);
63     deserialise(inputStream, instance.m_something);
64     deserialise(inputStream, instance.m_vecs);
65     return inputStream;
66 }
67
68 private:
69     int m_int;
70     float m_float;
```

```
71     std::string m_string;
72     Something2 m_something;
73     std::vector<Something2> m_vecs;
74 }
```

65.6 Next Steps

Now all you need to do is write serialisation specializations for the other STL containers you intend to use. Then, all your classes just need to have a serialisation macro added to them. You might have some cases when you’re playing with raw pointers, etc, in which case, you can write your own `operator<<` and `operator>>`, it will still be compatible with these ones, after all, they’re literally just resulting in those operators.

Also, have a think about doing this with something like JSON. It’s actually not too hard if you’re familiar with the format. For example, you already have the member names in the macro, those would be used for the keys.

Chapter 66

Technique: Take a `constexpr` string from Compile Time to Run Time

里缪 2023-04-08  ★★★

Constant Expressions 是 C++ 元编程进入第二个时期的标志，从 C++11-23，标准在不断完善这一基础设施，从而为进入下个核心时期做准备。

本文要讨论的，是该进程中的某个点——C++20 `constexpr std::string`。

C++20 之前，标准已经陆续增加不少`constexpr`特性，但还不支持容器（除了像`std::array`这样的静态容器）。主要问题就在于，没有解决编译期的动态内存分配。虽然 C++20 通过 `transient allocation` 初步解决了这个问题，但还是有一定的局限性。

这个局限性，就是本文要讨论的中心。

下面正式进入主题。

C++20 的`constexpr std::string`，并不像其他编译期变量那样，能够直接使用。

```
1 #include <string>
2
3 int main() {
4     constexpr std::string str = "compile time string";
5 }
```

这段代码是无法编译的，因为`std::string` 内部需要动态内存分配。我们只能将`std::string` 放在`constexpr/consteval`修饰的函数里面使用。

```
1 constexpr std::string make_string() {
2     std::string str{"compile time string"};
3     return str;
4 }
5
6 int main() {
7     static_assert(make_string() == "compile time string");
8 }
```

为什么这里就支持动态内存分配了呢？其实，这并非完全意义上的动态内存分配，这种分配叫做 **transient allocation**。顾名思义，这是一种瞬逝的分配，它允许在 **constexpr expression** 内分配内存，但是必须在 expression 结束时释放内存。编译器通过这种方式来跟踪内存分配，便于控制的同时也易于实现。

简单来说， transient allocation 所分配的内存，在编译期结束前必须释放。

C++20 的 transient allocation 只允许使用 `new` 和 `std::allocator::allocate`，C++23 增加了对 `std::unique_ptr` 的支持。一个小例子：

```

1 constexpr void g(int *p) {
2     delete[] p;
3 }
4
5 constexpr int f(unsigned int size) {
6     auto p = new int[size];
7     std::iota(p, p + size, 1);
8     auto retval = std::count_if(p, p + size, [](int val) { return val % 4 == 0; });
9     g(p); // delete p in g
10    return retval;
11 }
12
13 int main() {
14     constexpr auto val = f(9);
15     std::cout << val; // output: 2
16 }
```

transient allocation 有相关的检测机制，用来检查在 **constexpr expression** 的生命周期内，是否正确分配并释放了内存，像是忘记释放，或是 `delete` 与 `delete[]` 的错配，都能够编译期检测到。

假如我们注释掉第 9 行的调用，则会产生编译期错误。(fig. 66.1)

```

<source>: In function 'int main()':
<source>:16:26: error: 'f(9)' is not a constant expression because allocated storage has not been deallocated
  16 |     auto p = new int[size];
      |           ^
ASM generation compiler returned: 1

```

图 66.1: error

介绍完了 transient allocation，现在让我们回到 **constexpr std::string** 的正题中来。由于 transient allocation 的局限性，我们无法保存 **constexpr std::string** 的值。

```

1 constexpr std::string make_string() {
2     std::string str{"compile time string"};
3     return str;
4 }
5
6 int main() {
7     // static_assert(make_string() == "compile time string");
```

```

8     constexpr auto str = make_string(); // Error!
9 }
```

为什么呢？ transient allocation 的内存在编译期结束时就已经释放了，故而`make_string()`的输出不能在运行期使用。

此时就需要一种方式来将 `constexpr string` 的输出保存起来，以在运行期使用。总的来说，有两种解决思路。

第一种思路，将`constexpr string` 的输出保存到`constexpr array`里，因其没有用到 transient allocation，所以可以在运行期来用。

```

1  constexpr auto make_string(int n) {
2
3     std::string str;
4     std::array<char, 10> buf;
5     for (auto i : std::views::iota(0, n)) {
6
7         // Convert integers to strings with constexpr to_chars in C++23
8         if (auto [ptr, ec] = std::to_chars(buf.data(), buf.data() + buf.size(), i);
9             ec == std::errc())
10            str += std::string_view(buf.data(), ptr);
11    }
12
13    return str;
14
15 }
16
17 template <std::size_t Len>
18 constexpr auto get_array(std::string_view str) {
19     std::array<char, Len + 1> buf { 0 };
20     std::copy(str.begin(), str.end(), buf.begin());
21     return buf;
22 }
23
24
25 int main() {
26     static_assert(make_string(11) == "012345678910");
27     constexpr static auto length = get_length(make_string(11));
28     constexpr static auto buf = get_array<length>(make_string(11));
29     constexpr static auto str = std::string_view(buf.begin(), buf.size());
30     std::cout << str << "\n"; // Output: 012345678910
31 }
```

其中，`make_string`接受一个数值，然后将`[0, n)`的数值依次转换成字符串，再保存到`constexpr string`。整个函数都在编译期执行，因此`std::to_string`之类的转换函数都不可以用，这里使用的是 C++23 的`constexpr std::to_chars`来完成这个工作。

`constexpr std::array` 需要指定长度，再使用之前必须得到数据的长度，这个工作由`get_length` 函数完成。

然后，`get_array` 将`constexpr string` 的输出全部拷贝到`constexpr array` 中，如此一来，它的生命期就可以延长到运行期。也因此，我们最终才能够借其构造一个`string_view`，通过`cout` 来输出这个结果。

这种方式的缺点就是要构造两次数据，因为在调用`get_array` 之前，必须先调用一次数据来获取其长度。

第二种思路同样是要借助`constexpr std::array`，但是要消除获取长度这一步骤。消除手法就是先选择一个比较大的长度，得到一个不符合实际大小的`array`。

```

1 constexpr auto get_array_helper(std::string_view str) {
2     std::array<char, 10 * 1024 * 1024> buf;
3     std::copy(str.begin(), str.end(), buf.begin());
4     return std::make_pair(buf, str.size());
5 }
```

在这个函数里面，将实际长度保存起来，接着再来将这个过长的`array` 缩小到实际长度。这里返回值为一个`pair`，第一个值就是过长的`array`，第二个值是数据实际的长度。

然后再对这个过长的`array` 进行缩放。

```

1 constexpr auto get_array2(std::string_view str) {
2     // structured binding declaration cannot be constexpr.
3     // Error!
4     constexpr auto pair = get_array_helper(str);
5     constexpr auto buf = pair.first;
6     constexpr auto size = pair.second;
7     std::array<char, size> newbuf;
8     std::copy(buf.begin(), std::next(buf.begin(), size), newbuf.begin());
9     return newbuf;
10 }
```

但是注意，以上实现是存在问题的，编译器报如 (fig 66.2)。

```

<source>:48:47: error: 'str' is not a constant expression
  48 |       constexpr auto pair = get_array_helper(str);
                  ^

```

图 66.2: error

此时要采用一种方式将`str` 转换成 constant expression，一个技巧是通过`lambda` 传递，而非直接传递实际值。然后执行传递的`lambda`，就可以得到一个`constexpr value`。于是修改代码为：

```

1 template <class Callable>
2 constexpr auto get_array(Callable call) {
3     // structured binding declaration cannot be constexpr.
4     constexpr auto pair = get_array_helper(call());
```

```

5  constexpr auto buf = pair.first;
6  constexpr auto size = pair.second;
7  std::array<char, size> newbuf;
8  std::copy(buf.begin(), std::next(buf.begin(), size), newbuf.begin());
9  return newbuf;
10 }
```

之后的操作，就是根据已知的实际大小，重新构造一个新的数据，它是符合实际大小的。现在，就可以无需额外获取一次大小，再来调用`get_array`。

```

1 constexpr auto make_data = [] {
2     return make_string(11);
3 };
4
5 constexpr static auto data = get_array(make_data);
6 constexpr static auto str = std::string_view{data.begin(), data.size()};
7 std::cout << str << "\n"; // Output: 012345678910
```

但是我们依旧不能一步到位地得到一个`string_view`，总是要经过一个`get_array`，难免麻烦。因此，让我们再进一步，将所有内容整合起来，提供一个独立的函数。

```

1 constexpr auto to_string_view(auto callable) {
2     constexpr static auto data = get_array(callable);
3     return std::string_view{data.begin(), data.size()};
4 }
5
6 constexpr static auto str = to_string_view(make_data);
7 std::cout << str << "\n";
```

现在调用起来就显得非常简明了。

但是这个代码能够编译通过是令人比较惊讶的，因为 C++20 **constexpr function** 中是不允许使用`constexpr static`变量的，一查才知道，原来是 C++23 去除了这一限制。

那么在 C++20 如何实现这一目的呢？可以采用这个技巧。

```

1 template <auto Data>
2 constexpr const auto& make_it_static() {
3     return Data;
4 }
5
6 constexpr auto to_string_view(auto callable) {
7     constexpr auto& data = make_it_static<get_array(callable)>();
8     return std::string_view{data.begin(), data.size()};
9 }
```

NTTP 和 **constexpr** 变量具有一些相同的属性，模板在推导时会实例化出想要的字符串。(fig. 66.3)

```

template parameter object for std::array<char, 12ul>{char [12]{(char)48, (char)49, (char)50,
    .ascii "012345678910"
main::str:
    .quad 12
    .quad template_parameter_object_for_std::array<char, 12ul>{char [12]{(char)48, (cha

```

图 66.3: NTTP

通过这种方式，就能够达到和`constexpr static`相同的效果。

最后，也不做小结了，说说过程中遇到的一些问题。

clang 对`constexpr string` 的支持很不足，几乎用不了；msvc 的确支持`constexpr string`，但它在用`array` 初始化`string_view` 存在错误，导致无法将`get_array` 的结果转换成`string_view`。按理说 clang 和 msvc 的版本目前要比 gcc 新，问题应该更少，gcc13 都还没出来呢¹，但使用起来，还是 gcc 的问题更少一点，本文的代码就是在 gcc trunk 测试通过的。

本文中的代码细节极多，而且用到了一些 C++23 的新特性，必须手动跑一下才能理解其中的奥妙。

¹ 里缪注：4月26日 gcc13.1 已发布。

Part IX

Tricks

Techniques 和 Tricks 是两个比较相似的主题，前者侧重于灵活地使用语言特性来解决现有问题，后者则侧重于采用意想不到的方式来使用语言特性。

Tricks 往往过于聪明，谓之以奇技淫巧。历史上很多时候，由于缺少一些语言特性，某些问题必须使用这些奇技淫巧，某些问题则可以借其高效解决，可以说它们也推动了技术和标准的发展。

本 Part 包含了一些与其相关的文章，内容相对较短，难度等级普遍偏低，能让大家快速了解一些小技巧。

2023 年 5 月 14 日
水滴会飞

Chapter 67

如何优雅地打印类型名称？

里缪 2022-09-16



打印类型名称，听起来像是一个很简单的需求，但在目前的 C++ 当中，并非易事。

本文介绍了一些对此需求的分析与实现。

67.1 概述

类型属于 type，对象属于 value，前者是编译期的东西，后者则是运行期的东西。

你可以打印一个变量的值，却无法打印一个类型的名称。

那么如何才能实现这个需求？通常来说，解决问题的思路是将新问题转换为已经存在解决方案的旧问题。

其一，编译期目前只能输出错误信息，这个错误信息也可以是一种打印类型名称的方法。我们需要做的，就是主动触发报错，可以利用重载决议的相关知识达到这个目的。

其二，既然无法直接打印类型，那么就将类型转换为 value，从而在运行期进行打印。但是，通过表格暴力转换法其实并不可行，因为类型组合起来实在太多了。此时可以借助一些语言或编译器特性来获取到类型信息，比如通过 typeid 就可以根据类型得到一个简单的名称。

思路确定了，接着就可以顺着这个思路设计实现，以下各节展示各种实作法。

67.2 编译期打印类型名称

这种思路是利用错误信息输出类型信息，如何触发错误，如果大家已经读过【洞悉 C++ 函数重载决议】，相信已经有了深刻认识。

具体实现如下：

```
1 template <typename...> struct type_name {};
2 template <typename... Ts> struct name_of {
3     using X = typename type_name<Ts...>::name;
4 };
5
6 int main() {
```

```

7     name_of<int, float, const char*>();
8 }
```

由于 Name Lookup 查找的名称 name_of 带有模板，因此会进入重载决议的第二阶段：Template Handling。

模板参数已经显式指出，因此其实并不会进行 TAD，而是直接模板参数替换。但是编译器发现 type_name<Ts...>::name 并没有 name 类型，因此模板替换失败，产生 hard error，编译失败。

这个 hard error 错误信息，就带有类型名称，输出如下：

```

error: no type named 'name' in 'struct type_name<int, float, const char*>'
7 |     using X = typename type_name<Ts...>::name;
```

现在，就可以使用该实现在编译期查看实际类型的名称，比如：

```

1 template <typename T>
2 void f(T t) {
3     name_of<decltype(t)>();
4 }
5
6 int main() {
7     const int i = 1;
8     f(i);
9 }
```

输出为：

```

error: no type named 'name' in 'struct type_name<int>'
7 |     using X = typename type_name<Ts...>::name;
```

可以看到，TAD 在推导参数时丢弃了 top-level const 修饰，t 实际类型为 int。

这种形式的优点是完全发生于编译期，实现简单；缺点也很明显，无法指定输出形式，看起来不够直观。

67.3 Demangled Name

另一种方式是借助 typeid 关键字，通过它可以获得一个 std::type_info 对象，其结构如下。

```

1 namespace std {
2     class type_info {
3     public:
4         virtual ~type_info();
5         bool operator==(const type_info& rhs) const noexcept;
6         bool before(const type_info& rhs) const noexcept;
7         size_t hash_code() const noexcept;
8         const char* name() const noexcept;
9         type_info(const type_info&) = delete; // cannot be copied
10        type_info& operator=(const type_info&) = delete; // cannot be copied
```

```
11     };
12 }
```

其中的成员函数 name() 就可以返回类型的名称，这样就根据 type 获取到了 value。但是标准说这个名称是基于实现的。

Returns an implementation defined null-terminated character string containing the name of the type. No guarantees are given; in particular, the returned string can be identical for several types and change between invocations of the same program.

事实上也的确如此，MSVC 返回的是一段可读的类型名称，而 gcc, clang 返回的是 Mangled Name。（Name Mangling 内容可以参考【洞悉 C++ 函数重载决议】）如图67.1。

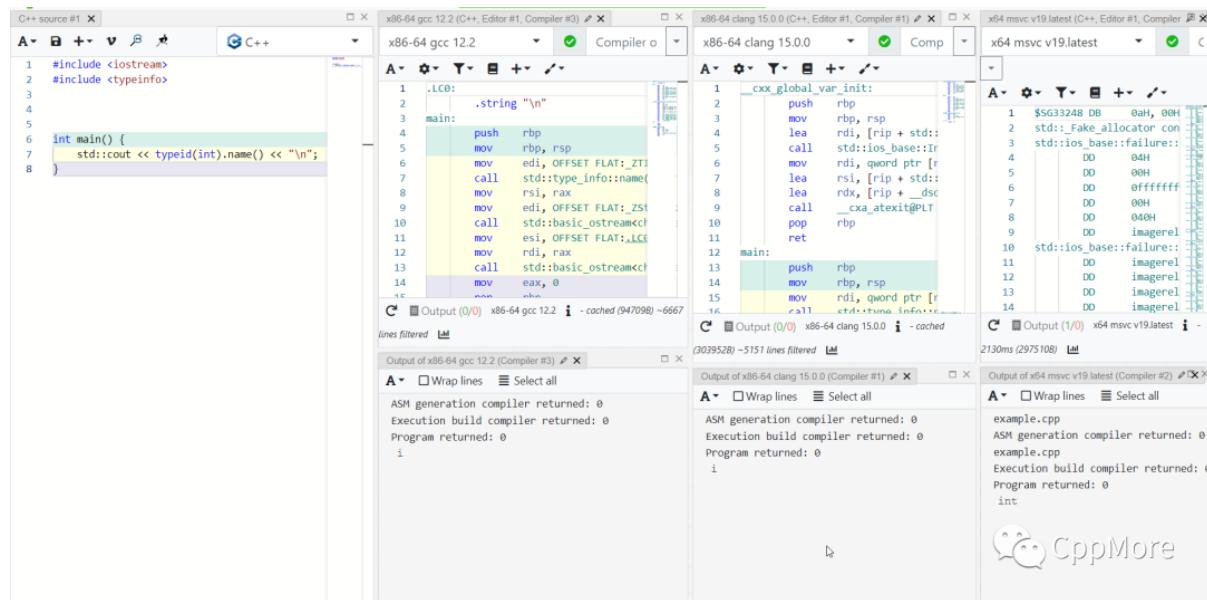


图 67.1: msvc, gcc, clang

但幸好，它们内部提供的有 Demangle API，通过相关 API 就可以将类型名称转换为可读的名称。这个 API 定义如下：

```
namespace abi {
    extern "C" char* __cxa_demangle (const char* mangled_name,
                                      char* buf,
                                      size_t* n,
                                      int* status);
}
```

这里主要关注第一个参数就可以，其他参数都可以置空。第一个参数就是 type_info::name() 返回的 Mangled Name，返回值为 Demangled Name。

因此，现在就可以分而论之，msvc 直接使用 type_info::name() 返回的类型名称就可以；对于 gcc/clang，则先使用 Demangle API 进行解析，次再使用。

具体实现如下：

```
1 #include <iostream>
2 #include <string>
```

```

3 #include <typeinfo>
4 #include <type_traits>
5 #ifndef _MSC_VER
6     #include <cxxabi.h>
7 #endif // _MSC_VER
8
9 template <typename T>
10 std::string type_name() {
11     using type = typename std::remove_reference<T>::type;
12
13     // 1. 通过 typeid 获得类型名称
14     const char* name = typeid(type).name();
15     std::string result;
16
17     // 2. 通过 gcc/clang 扩展 API 获得 Demangled Name
18 #ifndef _MSC_VER
19     char* demangled_name = abi::__cxa_demangle(name, nullptr, nullptr, nullptr);
20     result += demangled_name;
21     free(demangled_name);
22 #else
23     result += name;
24 #endif // _MSC_VER
25
26     // 3. 添加丢弃的修饰
27     if (std::is_const<type>::value)           result += " const";
28     if (std::is_volatile<type>::value)         result += " volatile";
29     if (std::is_lvalue_reference<T>::value)    result += "&";
30     if (std::is_rvalue_reference<T>::value)    result += "&&";
31
32     return result;
33 }
34
35 struct Base {};
36 struct Derived : Base {};
37
38 int main() {
39     std::cout << type_name<const int&>() << "\n";
40     std::cout << type_name<Base>() << "\n";
41     std::cout << type_name<Derived>() << "\n";
42 }
```

实现分为三个步骤，注释已经写得很清晰了，这里补充几个重点。

第一，Demangled API 的返回值是采用 malloc() 分配的内存，需要手动进行释放。

第二, `type_info::name()` 会丢弃 `cv` 及引用修饰符, 所以还需要手动添加这些修饰。

最终输出如图67.2所示。

```

x64 msvc v19.latest (C++, Editor #1, Compiler #1) x
x64 msvc v19.latest /s
1 $SG50957 DB ' const',
2 ORG $+1
3 $SG50959 DB ' volatile'
4 ORG $+2
5 $SG50961 DB '&, 00H

x86-64 clang 15.0.0 (C++, Editor #1, Compiler #2) x
x86-64 clang 15.0.0
1 __cxx_global_var_init:
2     push    rbp
3     mov     rbp, rsp
4     lea     rdi, [rip + 8]
5     call    std::ios_
6     .string "\n"

x86-64 gcc 12.2 (C++, Editor #1, Compiler #3) x
x86-64 gcc 12.2
1 .LC0:
2     .string "\n"
3 main:
4     push    rbp
5     mov     rbp, rsp

Output of x64 msvc v19.latest (Compiler #1) x
example.cpp
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
int const&
Base
Derived

Output of x86-64 clang 15.0.0 (Compiler #2) x
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
int const&
Base
Derived

Output of x86-64 gcc 12.2 (Compiler #3) x
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
int const&
Base
Derived

```

图 67.2: msvc, gcc, clang results

这种实现方式来自 <https://stackoverflow.com/a/20170989>。

优点在于, 可以统一格式, 输出清晰。缺点在于, 实现稍微麻烦, 要考虑更多情况, 且发生于运行期。

67.4 编译器扩展特性

编译器还存在另一种扩展, 包含有类型信息。大家也许用过 `__func__`, 这是每个函数内部都会预定义的一个标识符, 表示当前函数的名称。于 C99 添加到 C 标准, C++11 添加到了 C++ 标准, 定义如下。

```
static const char __func__[] = "function-name";
```

C++ 引入的这个说是“implementation-defined string”, 意思也是基于实现的, 不过在三个平台上的输出基本是一致的。

这个标识符只包含函数名称, 并不会附带模板参数信息。但是与其相关的扩展附带有这部分信息, `gcc/clang` 的扩展为 `__PRETTY_FUNCTION__`, `msvc` 的扩展为 `__FUNCSIG__`。

它们的内容形式也是基于实现的, 一个简单的例子如下。

```

1 template <typename T>
2 consteval auto type_name() {
3 #ifdef _MSC_VER
4     return __FUNCSIG__;

```

```

5 #elif defined(__GNUC__)
6     return __PRETTY_FUNCTION__;
7 #elif defined(__clang__)
8     return __PRETTY_FUNCTION__;
9 #endif
10 }
11
12 int main() {
13     std::cout << type_name<int>();
14 }
```

输出分别为：

```

1 // gcc
2 consteval auto type_name() [with T = int]
3
4 // clang
5 auto type_name() [T = int]
6
7 // msvc
8 auto __cdecl type_name<int>(void)
```

gcc 的这种格式不错，clang 丢弃了 consteval，msvc 同样如此，但它加上了函数调用约定。

现在需要做的，就是根据这些信息，解析出想要的信息。可以借助 C++17 std::string_view 在编译期完成这个工作。

具体实现如下。

```

1 template <typename T>
2 consteval auto type_name() {
3     std::string_view name, prefix, suffix;
4 #ifdef __clang__
5     name = __PRETTY_FUNCTION__;
6     prefix = "auto type_name() [T = ";
7     suffix = "]";
8 #elif defined(__GNUC__)
9     name = __PRETTY_FUNCTION__;
10    prefix = "consteval auto type_name() [with T = ";
11    suffix = "]";
12 #elif defined(_MSC_VER)
13    name = __FUNCSIG__;
14    prefix = "auto __cdecl type_name<";
15    suffix = ">(void)";
16 #endif
17    name.remove_prefix(prefix.size());
18    name.remove_suffix(suffix.size());
```

```

19
20     return name;
21 }
```

通过使用 `std::string_view`, 以上代码全都发生于编译期。该代码来自 <https://stackoverflow.com/a/56766138>。

这个实现方式要比 Demangled Name 好, 不会丢失修饰, 类型信息完善, 且发生于编译期。缺点也有, 编译器扩展一般都是基于实现的, 没有标准保证, 内容形式可能会改变, 依赖于此的实现并不具备较强的稳定性。

67.5 Circle

对比以上实现, 可以发现, 反而是第一种办法, 即主动触发 Name Lookup 报错这种方式最简单, 且最稳定、最通用。其他方法都依赖了编译器扩展特性, 虽然可以达到目的, 但技巧偏多, 没有保证。

大家要是读过之前更新的四章「C++ 反射」文章, 就知道类型名称其实是一个最基本的类型元信息, 只要编译器支持反射, 那么实现这个需求是再简单不过了。

在此, 我们就来看看 Circle 提供的强大元编程能力, 是如何优雅地实现这个功能的。注: Circle 基本内容, 请看 C++ 反射第三章。

Circle 对于该需求的实现如下:

```

1 template <typename... Ts>
2 void print_types() {
3     printf("%d - %s\n", int..., Ts.string)...;
4 }
5
6 print_types<int, double, const char*, int&&>();
7
8 // output:
9 // 0 - int
10 // 1 - double
11 // 2 - const char*
12 // 3 - int&&
```

是不是太简单了! 而且还要强大许多, 比如还可以去重、排序:

```

1 template <typename... Ts>
2 void f() {
3     printf("unique:\n");
4     print_types<Ts.unique...>();
5
6     printf("sort by type name: \n");
7     print_types<Ts.sort(_0.string < _1.string)...>();
8 }
9
10 f<int, double, const char*, double, int&&>();
```

```

11
12 // output:
13 // unique:
14 // 0 - int
15 // 1 - double
16 // 2 - const char*
17 // 3 - int&&
18 // sort by type name:
19 // 0 - const char*
20 // 1 - double
21 // 2 - double
22 // 3 - int
23 // 4 - int&&

```

这些操作只要加上 @meta 就全部可以发生于编译期。

所以说，自己的实现是完全比不上这种编译器自带的类型元信息机制的，因为解决问题时不在一个层次。用户自己实现的永远处在应用层，而编译器则直接在原理层建设。

67.6 Static Reflection

本节再说说如何使用 C++ 标准反射来实现该需求，就它目前的发展，还没有 Circle 的反射强，不过标准反射的「源码注入」能力很强。详情请看 C++ 反射第四章。

通过标准元函数 `name_of()` 就可以获取类型名称，因此实现其实很简单，代码如下。

```

1 template <typename T>
2 consteval auto type_name() {
3     return meta::name_of(reflexpr(T));
4 }
5
6 int main() {
7     const int i = 1;
8     constexpr auto __dummy = __reflect_print(type_name<decltype(i)>());
9 }

```

这里，将在编译期输出 `const int`。

虽然标准反射目前来说还是一个残缺品，但实现这种需求也比自己实现起来要简单太多了。

67.7 总结

本文不算太难，串着讲了一些东西，主要是当时研究 TAD 时写过相关工具，索性写一篇完整的文章。

很多时候，编译器推导的类型并不和预期一致，使用本文介绍的工具可以很方便地研究编译器的这些行为。

这里还串起了重载决议和反射的相关内容，也算是帮大家回顾一下。

Chapter 68

Avoiding direct syscall instructions by using trampolines

👤 eversinc33 📅 2022-08-30 💬 ★★

Recently, in order to prepare for an internal penetration testing engagement, I wanted to automate my payload generation. In order to do so, I created a packer for executables and shellcodes called MATROJKA. Since I've been a fan of Nim for malware development for some time, the choice to write my packer in Nim was an easy one. Nim has a beautiful syntax, transpiles to C, has great C and C++ (yes, real C++) integrations and is overall very fun to write in.

There are a few publicly available packers already that are based on Nim - most notably @chvancooten¹'s NimPackt-v1 packer and @icyguider²'s Nimcrypt2. NimPackt-v1 is a shellcode- and dotnet-assembly packer that is actively used by threat actors in the wild³. It's second version, NimPackt-NG improves upon v1, but is, as of now, still private. Nimcrypt2 is a packer for shellcode, dotnet-assemblies and additionally supports regular portable executables.

Both NimPackt-v1 and Nimcrypt2 use SysWhispers (as implemented by NimlineWhispers⁴) to invoke direct syscalls in order to avoid EDR-hooks. If you don't know what syscalls in Windows are, and what role they play in malware development and -detection, I can highly recommend this article by @Cn33liz⁵.

In short, syscalls are undocumented functions in **NTDLL.DLL** that are the closest to the kernel that we can get. In the end, any windows-API call calls one or more of these functions. If you for example call **WriteProcessMemory**, the syscall that is executed under the hood is **NtWriteVirtualMemory**. As such, AVs and EDRs like to hook some of these syscalls that are interesting to us malware developers, in order to inspect what is executed by a program. There are several methods to "unhook" these syscalls and make sure that our calls go to the kernel without being monitored - one example would be to reload a copy of **NTDLL.DLL** from disk, that is not yet hooked by the AV/EDR. Another is to find out the syscall numbers (which are different, depending on the Windows version) in some way and insert them into assembly stubs which we can be sure are not hooked, e.g. like this:

¹<https://github.com/chvancooten/NimPackt-v1>

²<https://github.com/icyguider/Nimcrypt2>

³<https://twitter.com/blackorbird/status/1553685027753365505>

⁴<https://github.com/ajpc500/NimlineWhispers>

⁵<https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/>

```

1 mov r10, rcx
2 mov eax, <SYSCALL_NUMBER_TO_INSERT>
3 # here would be where a hook would be inserted, that jumps to code that belongs to the AV/EDR
4 syscall
5 ret

```

This is what SysWhispers does - it generates assembly code that is then filled with the syscall numbers depending on the version of Windows that the code is running on.

68.1 Detecting direct syscalls made through SysWhispers

The problem with using SysWhispers1 and SysWhispers2 (not 3 though) is that by now it is heavily signatured by most AVs and EDRs and most binaries generated with tools that leverage SysWhispers are easily flagged as malicious (I assume this is why Nimcrypt2 additionally supports using GetSyscallStub instead of NimlineWhispers2).

One simple way to detect the use of syscalls generated by SysWhispers is to check for direct `syscall` instructions. Usually, each syscall goes through `NTDLL.DLL`, which acts as Windows' interface to kernel mode, so direct `syscall` instructions should (in theory) never occur and are highly suspicious.

```
eversinc33@debian:~/Documents/NimPackt-v1/output$ objdump --disassemble -M intel SeatbeltExecAssemblyNimPackt.exe | grep -i syscall
41349d: 0f 05          syscall
4134df: 0f 05          syscall
413701: 0f 05          syscall
```

As such, Defender instantly removes a binary that includes NimlineWhispers2 (if no further evasion is applied) upon downloading it onto a Windows host:

```
PS C:\Users\everinc33> copy //192.168.2.126/TMP/seatbelt_nimpackt.exe
copy : Operation did not complete successfully because the file contains a virus or potentially unwanted software.
At line:1 char:1
+ copy //192.168.2.126/TMP/seatbelt_nimpackt.exe
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Copy-Item], IOException
+ FullyQualifiedErrorId : System.IO.IOException,Microsoft.PowerShell.Commands.CopyItemCommand
```

That meant that I had to look for a different way to invoke direct syscalls for my packer. I did not want to use `GetSyscallStub`⁶, since it is used by Nimcrypt2 and I figured that using a different technique would make my packer's signatures more unique and thus less detectable.

68.2 Retrieve syscalls with HellsGate

Another technique that is widely used to retrieve syscall numbers, in order to invoke unhooked syscalls is HellsGate by @smelly_vx and @am0nsec. You basically traverse the `PEB` structure⁷, until you reach the module list, get `NTDLL.DLL`'s base address and then traverse its `Export Address Table` until you find the desired function. API Hashing⁸ is used to find the function name. Then, all that is left is to extract the syscall number from that function and you have everything you need to call that syscall directly, by using a syscall

⁶<https://github.com/S3cur3Th1sSh1t/NimGetSyscallStub/blob/main/GetSyscallStub.nim>

⁷<https://malwareandstuff.com/peb-where-magic-is-stored/>

⁸<https://www.ired.team/offensive-security/defense-evasion/windows-api-hashing-in-malware>

stub like above. You can read the paper at the Vx-Underground Github⁹, which explains it more in-depth. Luckily, zimawhit3¹⁰ already implemented HellsGate in Nim.

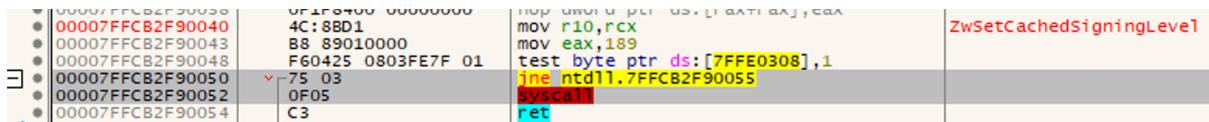
However, with HellsGate the same problem arises, since the assembly stubs that are populated with the retrieved syscall numbers also use the direct `syscall` instruction to invoke the syscall.

68.3 Make it bounce!

To make my syscalls seem more legit, I adjusted HellsGate, by simply replacing all `syscall` instructions with a trampoline jump - in this case a `JMP` instruction that jumps to the location of a `syscall` instruction located in `NTDLL.DLL`. This makes the syscalls seem more legit, as they originate from `NTDLL.DLL` and also avoids leaving any `syscall` instructions in the resulting binary. This technique is nothing new though, and was described e.g. in a blog post by @passthehashbrowns¹¹. In fact, I later found out that this is what SysWhispers3¹² and NimlineWhispers3¹³ ended up using as a remediation (well...). However, I still see it as a way to improve HellsGate. EDIT: I found out that this modification to HellsGate has already been done and is known as RecycledGate¹⁴

Thanks to Nim's ability to write inline assembly, implementing this was a breeze:

First, I parsed `NTDLL.DLL` byte by byte until a `syscall` instruction is found. In binary representation, the `syscall` instruction and its prologue are `0x75 0x03 0x0F 0x05`, as can be seen when inspecting the DLL in x64dbg:



Starting from the `NTDLL.DLL` module base address it doesn't take long for one to find such an address. We just take the first one we find and save it to the global variable `syscallJumpAddress`:

```

1 proc getSyscallInstructionAddress(ntdllModuleBaseAddr: PVOID): ByteAddress =
2     ## Get The address of a syscall instruction from ntdll to make sure
3     ## all syscalls go through ntdll
4     echo "[*] Resolving syscall..."
5     echo "[*] NTDLL Base: " & $cast[int](ntdllModuleBaseAddr).toHex
6     var offset: UINT = 0
7     while true:
8         var currByte = $cast[PDWORD](ntdllModuleBaseAddr + offset) []
9         if "050F0375" in $currByte.toHex:
10             echo "[*] Found syscall in ntdll addr "
11             & $cast[ByteAddress](ntdllModuleBaseAddr + offset).toHex
12             & ":" & $currByte.toHex

```

⁹<https://github.com/vxunderground/VXUG-Papers/tree/main/Hells%20Gate>

¹⁰<https://github.com/zimawhit3/HellsGateNim>

¹¹<https://passthehashbrowns.github.io/hiding-your-syscalls>

¹²https://klezvirus.github.io/RedTeaming/AV_Evasion/NoSysWhisper/

¹³<https://github.com/klezVirus/NimlineWhispers3>

¹⁴<https://github.com/theFLink/RecycledGate>

```

13     return cast[ByteAddress](ntdllModuleBaseAddr + offset) + sizeof(WORD)
14     offset = offset + 1

```

Now all that is left is to adjust the assembly code for each syscall and add a **JMP** to our address from above:

```

1 proc NtProtectVirtualMemory(
2     ProcessHandle: Handle,
3     BaseAddress: PVOID,
4     NumberOfBytesToProtect: PULONG,
5     NewAccessProtection: ULONG,
6     OldAccessProtection: PULONG): NTSTATUS { .asmNoStackFrame. } =
7     asm """
8         mov r10, rcx
9         mov eax, `ntProtectSyscall`
10        # syscall                      # this is what we want to avoid
11        mov r11, `syscallJumpAddress` # move syscall address into r11
12        jmp r11                      # jump to syscall address
13        ret
14 """

```

When compiling the binary, we do not have any direct syscalls left anymore, which resulted in a much smaller detection rate for my packed payloads. Neat!

```

eversinc33@debian:~/Documents/HellsGate-Trampoline$ objdump --disassemble -M intel hg.exe | grep syscall
eversinc33@debian:~/Documents/HellsGate-Trampoline$

```

Grabbing the first one available worked fine for me. One idea for improvement that is left is to control what **syscall** instruction we jump to and use this to e.g. fool an analyst that only observes the location of the syscall.

The code for this technique is hosted at <https://github.com/eversinc33/BouncyGate>. Unfortunately, as opposed to SysWhispers/NimlineWhispers, you will have to add the function definitions for each Syscall that you need yourself (but you can still use those that NimlineWhispers generates).

Happy Hacking!

Chapter 69

Avoiding Temporaries with Expression Templates

• morderncpp 📅 2022-04-08 🔖 ★

Expression templates are typically used in linear algebra and are "structures representing a computation at compile-time, which structures are evaluated only as needed to produce efficient code for the entire computation" (https://en.wikipedia.org/wiki/Expression_templates). In other words, expression templates are only evaluated when needed.

I provide you with this post only the key ideas of expression templates. To use them, you should study further content such as

- C++ Templates: The Complete Guide by David Vandervoorde, Nicolai M. Josuttis, and Douglas Gregor (<http://www.tmplbook.com/>)
- Boost Basic Linear Algebra Library (https://www.boost.org/doc/libs/1_59_0/libs/numeric/ublas/doc/index.html)
- Expression Templates Revisited by Klaus Iglberger (<https://www.youtube.com/watch?v=hfn0BVOegac>). Klaus's talk demystifies many performance-related myths about expression templates.

What problem do expression templates solve? Thanks to expression templates, you can get rid of superfluous temporary objects in expressions. What do I mean with superfluous temporary objects? My implementation of the class `MyVector`.

69.1 A first naive Approach

`MyVector` is a simple wrapper for a `std::vector<T>`. The wrapper has two constructors (lines 1 and 2), knows its length (line 3), and supports the reading (line 4) and writing (line 5) by index.

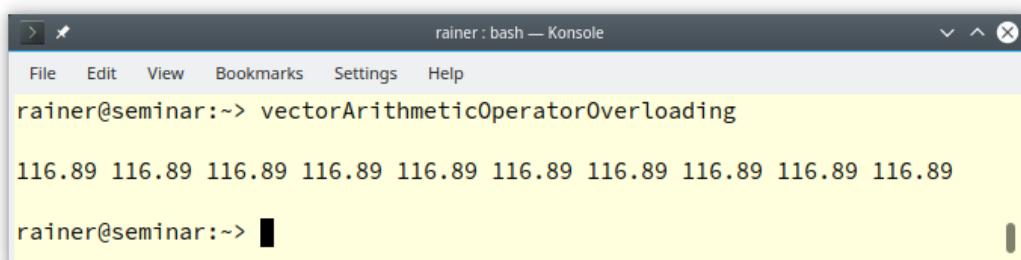
```
1 // vectorArithmeticOperatorOverloading.cpp
2
3 #include <iostream>
4 #include <vector>
5
```

```
6  template<typename T>
7  class MyVector{
8      std::vector<T> cont;
9
10 public:
11     // MyVector with initial size
12     MyVector(const std::size_t n) : cont(n){} // (1)
13
14     // MyVector with initial size and value
15     MyVector(const std::size_t n, const double initialValue)
16         : cont(n, initialValue){} // (2)
17
18     // size of underlying container
19     std::size_t size() const{ // (3)
20         return cont.size();
21     }
22
23     // index operators
24     T operator[](const std::size_t i) const{ // (4)
25         return cont[i];
26     }
27
28     T& operator[](const std::size_t i){ // (5)
29         return cont[i];
30     }
31
32 };
33
34 // function template for the + operator
35 template<typename T>
36 MyVector<T> operator+ (const MyVector<T>& a, const MyVector<T>& b) // (6)
37 {
38     MyVector<T> result(a.size());
39     for (std::size_t s = 0; s <= a.size(); ++s){
40         result[s] = a[s] + b[s];
41     }
42     return result;
43 }
44
45 // function template for the * operator
46 template<typename T>
47 MyVector<T> operator* (const MyVector<T>& a, const MyVector<T>& b) // (7)
```

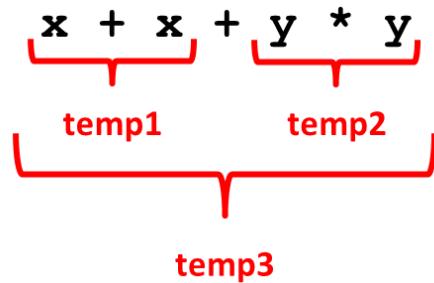
```

48 {
49     MyVector<T> result(a.size());
50     for (std::size_t s = 0; s <= a.size(); ++s){
51         result[s] = a[s] * b[s];
52     }
53     return result;
54 }
55
56 // function template for << operator
57 template<typename T>
58 std::ostream& operator<<(std::ostream& os, const MyVector<T>& cont) // (8)
59 {
60     std::cout << '\n';
61     for (int i = 0; i < cont.size(); ++i) {
62         os << cont[i] << ' ';
63     }
64     os << '\n';
65     return os;
66 }
67
68 int main(){
69
70     MyVector<double> x(10, 5.4);
71     MyVector<double> y(10, 10.3);
72
73     MyVector<double> result(10);
74
75     result = x + x + y * y;
76
77     std::cout << result << '\n';
78 }
79 }
```

Thanks to the overloaded `+` operator (line 6), the overloaded `*` operator (line 7), and the overloaded output operator (line 8) the objects `x`, `y`, and `result` behave like numbers.

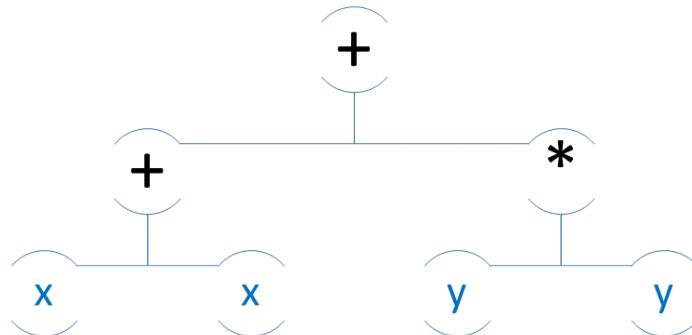


Why is this implementation naive? The answer is in the expression `result = x + x + y * y`. In order to evaluate the expression, three temporary objects are needed to hold the result of each arithmetic expression.



How can I get rid of the temporaries? The idea is simple. Instead of performing the vector operations greedy, I create the expression tree for `result[i]` at compile time lazily. Lazy evaluation means that an expression is only evaluated when needed.

69.2 Expression templates



There are no temporaries need for the expression `result[i] = x[i] + x[i] + y[i] * y[i]`. The assignment triggers the evaluation. Sad to say, but the cppinlive is even in this simple usage not so easy to digest.

```

1 // vectorArithmeticExpressionTemplates.cpp
2
3 #include <cassert>
4 #include <iostream>
5 #include <vector>
6
7 template<typename T, typename Cont= std::vector<T> >
8 class MyVector{
9     Cont cont;
10
11 public:
12     // MyVector with initial size

```

```
13  MyVector(const std::size_t n) : cont(n){}
14
15 // MyVector with initial size and value
16 MyVector(const std::size_t n, const double initialValue)
17     : cont(n, initialValue){}
18
19 // Constructor for underlying container
20 MyVector(const Cont& other) : cont(other){}
21
22 // assignment operator for MyVector of different type
23 template<typename T2, typename R2> // (3)
24 MyVector& operator=(const MyVector<T2, R2>& other){
25     assert(size() == other.size());
26     for (std::size_t i = 0; i < cont.size(); ++i)
27         cont[i] = other[i];
28     return *this;
29 }
30
31 // size of underlying container
32 std::size_t size() const{
33     return cont.size();
34 }
35
36 // index operators
37 T operator[](const std::size_t i) const{
38     return cont[i];
39 }
40
41 T& operator[](const std::size_t i){
42     return cont[i];
43 }
44
45 // returns the underlying data
46 const Cont& data() const{
47     return cont;
48 }
49
50 Cont& data(){
51     return cont;
52 }
53 };
54
```

```
55 // MyVector + MyVector
56 template<typename T, typename Op1 , typename Op2>
57 class MyVectorAdd{
58     const Op1& op1;
59     const Op2& op2;
60
61 public:
62     MyVectorAdd(const Op1& a, const Op2& b): op1(a), op2(b){}
63
64     T operator[](const std::size_t i) const{
65         return op1[i] + op2[i];
66     }
67
68     std::size_t size() const{
69         return op1.size();
70     }
71 };
72
73 // elementwise MyVector * MyVector
74 template< typename T, typename Op1 , typename Op2 >
75 class MyVectorMul {
76     const Op1& op1;
77     const Op2& op2;
78
79 public:
80     MyVectorMul(const Op1& a, const Op2& b ): op1(a), op2(b){}
81
82     T operator[](const std::size_t i) const{
83         return op1[i] * op2[i];
84     }
85
86     std::size_t size() const{
87         return op1.size();
88     }
89 };
90
91 // function template for the + operator
92 template<typename T, typename R1, typename R2>
93 MyVector<T, MyVectorAdd<T, R1, R2> >
94 operator+ (const MyVector<T, R1>& a, const MyVector<T, R2>& b)
95 {
96     return MyVector<T, MyVectorAdd<T, R1, R2> >(
```

```

97     MyVectorAdd<T, R1, R2 >(a.data(), b.data()));      // (1)
98 }
99
100 // function template for the * operator
101 template<typename T, typename R1, typename R2>
102 MyVector<T, MyVectorMul< T, R1, R2>>
103 operator* (const MyVector<T, R1>& a, const MyVector<T, R2>& b)
104 {
105     return MyVector<T, MyVectorMul< T, R1, R2>>(
106         MyVectorMul<T, R1, R2 >(a.data(), b.data()));    // (2)
107 }
108
109 // function template for < operator
110 template<typename T>
111 std::ostream& operator<<(std::ostream& os, const MyVector<T>& cont)
112 {
113     std::cout << '\n';
114     for (int i = 0; i < cont.size(); ++i) {
115         os << cont[i] << ' ';
116     }
117     os << '\n';
118     return os;
119 }
120
121 int main(){
122
123     MyVector<double> x(10,5.4);
124     MyVector<double> y(10,10.3);
125
126     MyVector<double> result(10);
127
128     result= x + x + y * y;
129
130     std::cout << result << '\n';
131
132 }
```

The key difference between the first naive implementation and this implementation with expression templates is that the overloaded + and + operators return in the case of the expression tree proxy objects. These proxies represent the expression trees (lines 1 and 2). The expression trees are only created but not evaluated. Lazy, of course. The assignment operator (line 3) triggers the evaluation of the expression tree that needs no temporaries.

The result is the same.

```
rainer@seminar:~> vectorArithmeticExpressionTemplates
116.89 116.89 116.89 116.89 116.89 116.89 116.89 116.89 116.89 116.89 116.89
rainer@seminar:~>
```

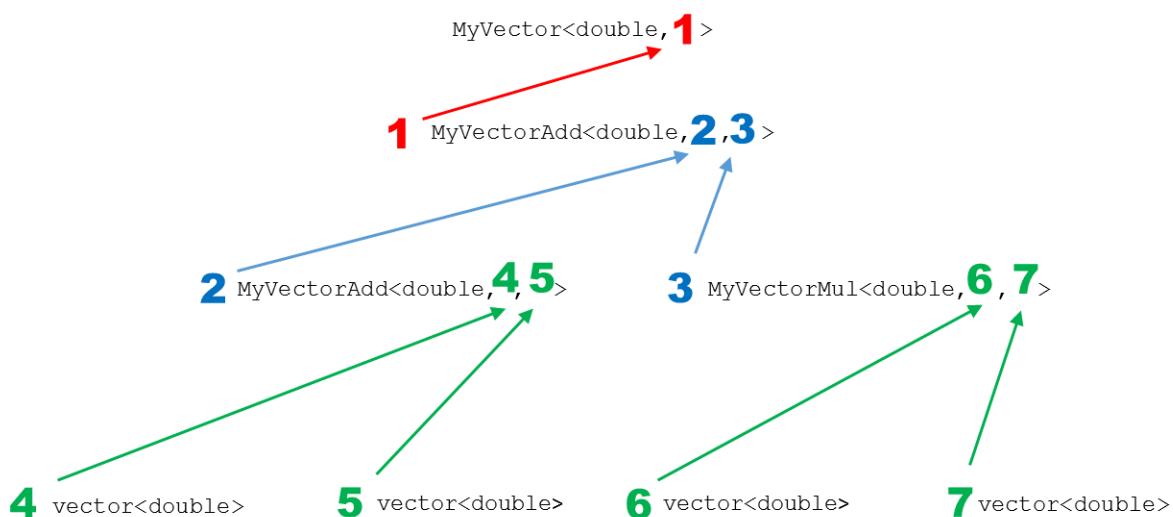
Thanks to the compiler explorer, I can visualize the magic of the program `vectorArithmeticExpressionTemplates.cpp`.

69.3 Under the hood

Here are the essential assembler instructions for the final assignment in the main function: `result= x + x + y * y.`

```
57      lea    rax, [rbp-100]
58      mov    rsi, rdx
59      mov    rdi, rax
60      call   MyVector<double, std::vector<double, std::allocator<double> > & MyVector<double,
           std::vector<double, std::allocator<double> >::operator=<double, MyVectorAdd<double, MyVectorAdd<double,
           std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double> > >,
           MyVectorMul<double, std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double>
           > > > (MyVector<double, std::vector<double, std::allocator<double> > >, MyVectorAdd<double, MyVectorAdd<double,
           std::vector<double, std::allocator<double> > >, MyVectorMul<double, std::vector<double,
           std::allocator<double> >, std::vector<double, std::allocator<double> > > > > const&)
61      lea    rax, [rbp-160]
62      mov    rdi, rax
63      call   void print<MyVector<double, std::vector<double, std::allocator<double> > > > (MyVector<double,
```

The expression tree in the assembler snippet looks quite scary, but with a sharp eye, you can see the structure. For simplicity reasons, I ignored `std::allocator` in my graphic.



Chapter 70

Casting a negative float to an unsigned int

👤 Burkhard Stubert 📅 2013-08-25 💬 ★★

```
1 auto fValue = -176.0;
2 auto tValue = static_cast<uint16_t>(fValue);
3 // On Intel: tValue = 65360
4 // On ARM:    tValue = 0
```

Never cast a negative `float` to an `unsigned int`. The result of the cast differs depending on whether the cast is executed on a device with Intel or ARM architecture. The C++ Standard decrees that the result of such a cast is undefined.

70.1 Introduction

I wrote the first version of this post in August 2013. It has been in the top 10 of my most popular posts ever since with 1550+ views per year. In November 2022, after 9 years, I decided to share the post on LinkedIn again. It garnered more than 100K impressions, 850 reactions, 65 comments and 25 reposts within a week. The blog post got its yearly number of views within a week.

It seems that the post hasn't lost its usefulness over the years and can still save people a lot of time. From some comments, I infer that I should give some context how the cast became undefined¹, a new section. The cast was perfectly fine, until someone violated the rules. The sections Why the Cast Fails² and How to Fix the Problem³ are taken from the original post with some minor changes. The post ends with the new section Earlier Feedback Needed⁴. The tooling support for detecting undefined behaviour is wanting.

¹<https://embeddeduse.com/2013/08/25/casting-a-negative-float-to-an-unsigned-int/#cast-became-undefined>

²<https://embeddeduse.com/2013/08/25/casting-a-negative-float-to-an-unsigned-int/#why-cast-fails>

³<https://embeddeduse.com/2013/08/25/casting-a-negative-float-to-an-unsigned-int/#how-to-fix>

⁴<https://embeddeduse.com/2013/08/25/casting-a-negative-float-to-an-unsigned-int/#earlier-feedback>

70.2 How the Cast Became Undefined

The ECUs (electronic control units) in harvesters, tractors, excavators, cranes and trucks communicate over CAN bus using the J1939 protocol. The J1939 standard⁵ demands that all numbers – integer or floating point numbers – are converted into unsigned integers for transmission over CAN. The ranges of the transformed values must fit into a 32-bit unsigned integer.

Let me illustrate this with an example. We assume that the outside temperature is in the range from -45°C to +85°C, because most electronics stops working outside this range. We measure the temperature with an accuracy of 0.1°C. We apply an affine transformation to convert the temperature -27.4°C into a non-negative floating-point number 176.0. The type of `fValue` and hence of `tValue` could also be `double`. It doesn’t make a difference for the cast.

```
float fValue;
auto tValue = (fValue + 45.0) * (1.0 / 0.1);
# With fValue = -27.4:
# tValue = (-27.4 + 45.0) * (1.0 / 0.1) = 17.6 * 10.0 = 176.0
```

We could now round `tValue`, take the ceiling or floor of it, or cast it to turn it into an unsigned integer. As this conversion may be executed hundreds or even thousands of times per second, casting seemed to be the fastest method. The other methods must execute some extra steps. The cppinline looks like this now. `tValue` is a 16-bit unsigned integer.

```
float fValue;
auto tValue = static_cast<uint16_t>((fValue + 45.0) * 10.0);
```

The behaviour of the cppinline is well defined. The range for the transformed values runs from 0 to 1300. J1939 messages would pack the values into 11 bits of its 8-byte payload. This saves 21 bits over a 4-byte `float` and 53 bits over an 8-byte `double`. Saving space is paramount, because CAN buses typically run at 256Kbps.

Now the inevitable happens. Someone flouts the J1939 standard, drops the offset from the parameter specification (Note: Conversion cppinline is generated!), and “just” uses a signed integer.

```
float fValue;
auto tValue = static_cast<uint16_t>(fValue * 10.0);
```

The unit tests for negative values pass on the PCs, which are still typically powered by Intel processors. Disaster strikes on the ARM-based ECUs. All negative values are cast to 0.

70.3 Why the Cast Fails

When we run the cppinline

```
1 auto fValue = -176.0;
2 auto tValue = static_cast<uint16_t>(fValue);
3 // On Intel: tValue = 65360
4 // On ARM:    tValue = 0
```

⁵<https://embeddeduse.com/2020/01/17/introduction-to-the-sae-j1939-standard/>

on a device with Intel architecture, `tValue` has the value 65360, which is equal to $2^{16} - 176$. 65360 is the two's complement representation of -176 for 16 bits and is the expected result.

When we run the `cppinline` on a device with ARM architecture, however, `tValue` has the value 0. Actually, every negative floating point number less than or equal to -1 comes out as 0. By the way, the result is the same if we remove the `static_cast`. Then, the conversion is done implicitly.

A post at StackOverflow⁶ points us to *Section 6.3.1.4 Real floating and integer* of the C Standard for an explanation to our problem:

The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type.

Thus, the range of portable real floating values is $(-1, \text{Utype_MAX}+1)$.

In other words, the two's complement of a negative float need not be computed. The safe range for casting a negative float to an unsigned int is from -1 to the maximum number that can be represented by this unsigned integer type.

70.4 How to Fix the Problem

A possible solution of our problem is to cast the floating point number to a signed integer number first and then to an unsigned integer number.

```

1 auto fValue = -176.0;
2 auto tValue = static_cast<uint16_t>(static_cast<int16_t>(fValue));
3 // On Intel: tValue = 65360
4 // On ARM:    tValue = 0

```

Another possible solution is to round the floating point number or to compute the floor or ceiling of the floating point number. Qt provides the functions `qRound`, `qCeil` and `qFloor` for this purpose. The right solution depends on the concrete problem at hand.

Unfortunately, the compiler cannot help us with finding the problem, because we use an explicit cast here. We basically tell the compiler that we know better what to do. If we had used an implicit conversion and had used the warning option `-Wconversion`, the compiler would have warned about the problem sheepishly.

```

1 auto fValue = -176.0;
2 uint16_t tValue = fValue;
3 warning: conversion from `float' to `quint16' {aka `short unsigned int'}
4     may change value [-Wfloat-conversion]

```

70.5 Earlier Feedback Needed

How can we detect the problem automatically?

- The Continuous Delivery (CD) pipeline runs the unit tests not only on the Intel-based workstation but also on the ARM-based target device.

⁶<http://stackoverflow.com/a/4752947/2277799>

- The CD pipeline runs the undefined behaviour sanitizer UBSan⁷.

The first method tells us that there is a problem and roughly where it is. We must still figure out that the problem is caused by the undefined cast. We may do this by searching the Web or by running several sanitizers. When we understand the problem, we add UBSan to the CD pipeline to avoid a regression.

So far, we have been talking about an ideal world. It's December 2022 and I haven't encountered an embedded software project yet that runs a CD pipeline. At least, I have helped a customer this year to set up a CD pipeline.

So, we must rely on people's expertise (knowing the problem or finding it quickly) and diligence (running tests manually on an ARM-based device). We get feedback very late, days or weeks after we wrote the problematic lines of `cppinline`. The CD pipeline is a big improvement. We get feedback within hours if not minutes. But the C++ tooling could do much better than this!

My favourite solution would be for the compiler to stop with an error when encountering undefined behaviour. That would be the earliest feedback possible.

```
error: Casting from a negative 'float' to an unsigned integer  
'uint16_t' is undefined behavior.
```

The compiler writers know that this cast is undefined, because the C++ standard says so. They know, because they implemented it differently for the same compiler, `g++`, on different processor architectures. The compiler behaves inconsistent.

Being inconsistent is one of the big no-gos in user experience (UX) design. Stopping with an error when encountering undefined behaviour would improve the UX of C++ compilers tremendously and save C++ developers lots of time.

Interpreting "undefined behaviour" as "the compiler can do whatever it wants" leads to a major waste of time for C++ developers. The C++ standard and compilers must treat undefined behaviour consistently as errors.

⁷<https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>

Chapter 71

Compiler Tricks to Avoid ABI-Induced Crashes

• brucedawson 📅 2022-12-14 🔍 ★★★

Last month I wrote¹ about an odd crash² that was hitting a few Chrome users. Something was corrupting the XMM7 register and that was causing Chrome to crash. We fixed a couple of bugs in Chrome and we were able to contact the third-party company whose software was causing the problems. They released a fixed version, and I assumed that my work was done.

```
task_source = task_tracker_->RunAndPopNextTask(std::move(task_source));

// Alias pointer for investigation of memory corruption
TaskSource* task_source_before_move = task_source.get();
base::debug::Alias(&task_source_before_move);

delegate_->DidProcessTask(std::move(task_source));

// Check that task_source is always cleared, to help detect
// corruption where task_source is non-null after being moved
// crbug.com/1218384
CHECK(!task_source);
```

图 71.1: Screen shot

However, instead of a gradual decline in the rates of this crash I saw a gradual increase. Apparently enterprise software updates roll out extremely slowly, and users were installing the old buggy version faster than they were updating to the fixed version. This situation will resolve itself eventually, but a lot of crashes were going to happen in the meantime. With the proper fix moving slowly through the pipelines I decided to try to hack in a decidedly improper fix.

The crashes happened because the compiler generated code that zeroed XMM7 and then expected (reasonably enough) that the register would retain its value. I had to convince the compiler to behave differently.

The problematic line of code was this one

¹<https://randomascii.wordpress.com/2022/11/21/please-restore-our-registers-when-youre-done-with-them/>

²<https://bugs.chromium.org/p/chromium/issues/detail?id=1218384>

```
delegate_->DidProcessTask(std::move(task_source));
```

The `std::move` was supposed to move the pointer in `task_source` to the function parameter, and zero out `task_source`. The zeroing was being done with XMM7 and was failing when XMM7 was corrupted.

71.1 Attempt #1: NOINLINE

As it turns out the moving and zeroing was actually happening in the `RegisteredTaskSource` move constructor. This was a separate function in a separate source file so normally it would not be able to make assumptions about the value of XMM7, but in our highly-optimized official builds we do Link Time Optimization (LTO)³ which can do cross-module inlining, so the body of the constructor was inserted into `RunWorker` which then allowed it to make these assumptions about XMM7. If I tagged this function as `NOINLINE` (shown in this candidate change list (CL)⁴) then the assumption would go away, and with it the crash.

I created a CL that did this, and looked at the generated code in our official builds and realized that there was another function that was inlined that was making the same assumption about XMM7. I tagged *that* one as `NOINLINE` and looked at the generated code again and found *another* function that was inlined that was making the same assumption about XMM7.

Playing whack-a-mole is not a great strategy for controlling the compiler. Even if I coerced the compiler into not depending on the XMM registers retaining their values the fix would not be robust. Any little change to the source code or the compiler could cause the bug to return, with arbitrarily bad memory corruption possibilities.

71.2 Attempt #2: optimize off

My next attempt was using a larger and more robust hammer. By telling the compiler to not optimize the `RunWorker` function at all I could be pretty certain that it wouldn't use any XMM registers. This method worked fine, and while shipping non-optimized code is normally undesirable it would have been fine in this case. The function in question was not performance sensitive and it would have worked. I wrote up an enormous comment block to explain why I was doing this strange and wonky thing and sent the change out for review⁵.

71.3 Attempt #3: it's clobbering time

Code-review can be a great process. It helps to maintain high code quality, and it gives other software developers a chance to offer suggestions. And they did. One code reviewer told me how to get what I wanted, and the other code reviewer told me where to put it.

One developer suggested that there might be a better option than “optimize off”. The root cause of the crashes was that XMM7 (and probably other XMM registers) were being corrupted. What if we told the compiler that this was happening? It turns out that the `gcc/clang` compilers offer a syntax for doing exactly that. These compilers allow inline assembly language, and in order for the inline code to coexist

³<https://clang.llvm.org/docs/ThinLTO.html>

⁴<https://chromium-review.googlesource.com/c/chromium/src/+/4087156>

⁵https://chromium-review.googlesource.com/c/chromium/src/+/4087650/1/base/task/thread_pool/worker_thread.cc

with the surrounding C/C++ code the `asm` statement includes an optional “clobbers” section⁶. This is a list of registers that the assembly language may have written to (used to fix Chrome’s own XMM7 clobbering⁷), and this is exactly what was needed. All I needed was this inline assembly block:

```
asm("" : : "%xmm6", "%xmm7", "%xmm8", "%xmm9", "%xmm10", "%xmm11",
     "%xmm12", "%xmm13", "%xmm14", "%xmm15");
```

This cryptic syntax says that the (empty) inline assembly block may have modified the XMM registers from XMM6 to XMM15 and that therefore the compiler should not assume anything about their values. Placing this right after the call to the arbitrary tasks ensures that the compiler will make no assumption about these registers. This is almost a perfect match for the problem.

The other developer who reviewed the change suggested that I move the magic `asm` block to a different location (`base::TaskAnnotator::RunTaskImpl`⁸). The original function that was crashing was just one place where tasks (that might clobber XMM registers) were called, and he told me where to put the `asm` block so that it would follow shortly after all calls to the potentially problematic tasks. This new location would prevent the crashes that we were seeing, and possible future crashes that might happen elsewhere.

71.4 Clobber blocks

I looked at the assembly language generated by various versions of my fixes in order to ensure that I was getting the desired results. The easiest way to do this was to load `chrome.dll` into `windbg` and then disassemble the problematic function. This command-prompt and then `windbg` set of commands does the trick:

```
> windbg -z chrome.dll
0:000> uf chrome!base::internal::WorkerThread::RunWorker
```

With the `NOINLINE` solution I could see fewer uses of XMM registers that were assumed to still be zero, but I did not get it to zero. With the `NOOPT` solution I saw that all use of XMM registers went away, although the corrupt values would be retained forever. With the `asm` block solution (initially tested in the `RunWorker` function, and then moved) I could see two changes:

One change was that the pattern of zeroing an XMM register before the loop and then using it in the loop went away. That makes sense because the `asm` directive explicitly told the compiler that that was not going to work.

The other change was that the `RunWorker` function started preserving the registers from XMM6 to XMM15 in the function prologue, and then restoring them in the epilogue. This makes sense because the clobber entries in the `asm` block tell the compiler that the XMM registers were clobbered, and the compiler is in charge of following the platform ABI. That is, the compiler needs to ensure that the functions that call

⁶<https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>

⁷https://chromium-review.googlesource.com/c/libyuv/libyuv/+/3972137/3/source/scale_gcc.cc

⁸https://chromium-review.googlesource.com/c/chromium/src/+/4087650/9/base/task/common/task_annotator.cc

```

0:000> uf 00000001`83f62a70
chrome!base::TaskAnnotator::RunTaskImpl [C:\b\vv\cache\builder\src\base\tas
115 00000001`83f62a70 4157 push    r15
115 00000001`83f62a72 4156 push    r14
115 00000001`83f62a74 4155 push    r13
115 00000001`83f62a76 4154 push    r12
115 00000001`83f62a78 56 push    rsi
115 00000001`83f62a79 57 push    rdi
115 00000001`83f62a7a 53 push    rbx
115 00000001`83f62a7b 4881ec10010000 sub     rsp, 110h
115 00000001`83f62a82 440f29bc2400010000 movaps  xmmword ptr [rsp+100h],xmm15
115 00000001`83f62a8b 440f29b424f0000000 movaps  xmmword ptr [rsp+0F0h],xmm14
115 00000001`83f62a94 440f29ac24e0000000 movaps  xmmword ptr [rsp+0E0h],xmm13
115 00000001`83f62a9d 440f29a424d0000000 movaps  xmmword ptr [rsp+0D0h],xmm12
115 00000001`83f62aa6 440f299c24c0000000 movaps  xmmword ptr [rsp+0C0h],xmm11
115 00000001`83f62aaef 440f299424b0000000 movaps  xmmword ptr [rsp+0B0h],xmm10
115 00000001`83f62ab8 440f298c24a0000000 movaps  xmmword ptr [rsp+0A0h],xmm9
115 00000001`83f62ac1 440f29842490000000 movaps  xmmword ptr [rsp+90h],xmm8
115 00000001`83f62aca 0f29bc24800000000 movaps  xmmword ptr [rsp+80h],xmm7
115 00000001`83f62add 0f29742470 movaps  xmmword ptr [rsp+70h],xmm6
115 00000001`83f62ad7 4989d6 nov     r14,rdx
115 00000001`83f62ada 488b0557d53908 nov     rax,qword ptr [chrome!__securit
115 00000001`83f62ad9 488b0557d53908 nov     rax,qword ptr [chrome!__securit

```

图 71.2: Disassembly of RunTaskImpl from windbg

RunWorker don't see registers being corrupted. So, once the fix was moved to RunTaskImpl⁹ the RunTaskImpl function would preserve and restore the non-volatile XMM registers and any calling functions would no longer see register corruption. Perfect!

71.5 Fixed, but should we?

After a few days of the **asm** block hack shipping to Chrome I can see that it has completely worked around the bug. Crashes in the canary (daily builds) channel have gone to zero even as the crash rate for the regular builds continue to climb. So, all good, right?

It's complicated. I'm pleased that we were able to address a pain point for our customers, but working around third-party bugs is *not* something that we want to get in the habit of doing. Doing so creates perverse incentives. As a general policy Chromium does not add patches to work around third-party bugs. We do not support having third-party code injected into our processes and we do not "fix" the crashes which that causes because that road leads to madness. As a general rule, if some third-party code (very often security software) is causing Chrome to crash then users should request a fix from their security vendor, should uninstall that software, or should configure it to not invade Chrome's processes.

I decided to push for this particular fix¹⁰ for a few reasons. The main reason was that the vendor has already pushed a fix. That is crucial. This **asm** hack is just a temporary workaround until their users install the updates. The secondary reason is that there was no obvious way for our mutual users to realize that Trellix disk encryption was the problem. The crashes happened even when no Trellix modules were injected into our processes -I don't know how -which meant Chrome was going to get blamed, with no easy way for us to explain ourselves.

I hope that this was the correct decision, and I hope that we are able to remove the clever-but-horrible **asm** hack after not too long.

⁹https://chromium-review.googlesource.com/c/chromium/src/+/4087650/9/base/task/common/task_annotator.cc

¹⁰<https://chromium-review.googlesource.com/c/chromium/src/+/4087650>

Chapter 72

Five tricky topics for data members in C++20

▀ Cpp Stories 2022-07-18 🔖 ★

Working with data members and class design is essential to almost any project in C++. In this article, I gathered five topics that I hope will get you curious about the internals of C++.

72.1 1. Changing status of aggregates

Intuitively a simple class type, or an array should be treated as “aggregate” type. This means that we can initialize it with braces {}:

```
1 #include <iostream>
2 #include <array>
3 #include <type_traits>
4 #include <utility>
5 #include <tuple>
6
7 struct Point {
8     double x {0.0};
9     double y {0.0};
10 };
11
12 int main() {
13     std::array<int, 4> numbers { 1, 2, 3, 4 };
14     std::array statuses { "error", "warning", "ok" }; // CTAD
15     Point pt { 100.0, 100.0 };
16     std::pair vals { "hello", 10.5f };
17     std::tuple pack { 10, true, "important" };
18
19     static_assert(std::is_aggregate_v<decltype(numbers)>);
```

```

20     static_assert(std::is_aggregate_v<decltype(statuses)>);
21     static_assert(std::is_aggregate_v<decltype(pt)>);
22     // not an aggregate...
23     static_assert(!std::is_aggregate_v<decltype(vals)>);
24     static_assert(!std::is_aggregate_v<decltype(pack)>);
25 }
```

Run @Compiler Explorer¹

But what is a simple class type? Over the years, the definition changed a bit in C++.

Currently, as of C++20, we have the following definition:

From latest C++20 draft dcl.init.aggr²:

An aggregate is an array or a class with

- no user-declared or inherited constructors,
- no private or protected direct non-static data members,
- no virtual functions and
- no virtual, private, or protected base classes.

However, for example, until C++14, non-static data member initializers (NSDMI or in-class member init) were prohibited. In C++11, the Point class from the previous example wasn't an aggregate, but it is since C++14.

C++17 enabled base classes, along with extended brace support. You can now reuse some handy aggregates as your base classes without the need to write constructors:

```

1 #include <string>
2 #include <type_traits>
3
4 enum class EventType { Err, Warning, Ok};
5
6 struct Event {
7     EventType evt;
8 };
9
10 struct DataEvent : Event {
11     std::string msg;
12 };
13
14 int main() {
15     DataEvent hello { EventType::Ok, "hello world" };
16
17     static_assert(std::is_aggregate_v<decltype(hello)>);
18 }
```

¹<https://godbolt.org/z/jvj4c8Tqv>

²<https://timsong-cpp.github.io/cppwp/n4868/dcl.init.aggr#:aggregate>

Run @Compiler Explorer³

If you compile with the std=c++14 flag, you'll get:

```
no matching constructor for initialization of 'DataEvent'
DataEvent hello { EventType::Ok, "hello world"};
```

Run at <https://godbolt.org/z/8oK1ree7r>

We also have some more minor changes like:

- user-declared constructor vs user-defined or explicit,
- inherited constructors

See more at:

- Aggregate initialization - [cppreference.com⁴](https://cppreference.com/w/cpp/language/aggregate_initialization)
- What are Aggregates and PODs, and how/why are they special? - Stack Overflow⁵

72.2 2. No parens for direct initialization and NSDMI

Let's take a simple class with a default member set to "empty" :

```
class DataPacket {
    std::string data_ {"empty"};
    // ... the rest...
```

What if I want data_ to be initialized with 40 stars *? I can write the long string or use one of the std::string constructors taking a count and a character. Yet, because of a constructor with the std::initializer_list in std::string which takes precedence, you need to use direct initialization with parens to call the correct version::

```
1 #include <iostream>
2
3 int main() {
4     std::string stars(40, '*');      // parens
5     std::string moreStars{40, '*'}; // <<
6     std::cout << stars << '\n';
7     std::cout << moreStars << '\n';
8 }
```

Run @Compiler Explorer⁶

If you run the code, you'll see:

```
*****
```

```
(*
```

³<https://godbolt.org/z/j8c84bdYo>

⁴https://en.cppreference.com/w/cpp/language/aggregate_initialization

⁵<https://stackoverflow.com/questions/4178175/what-are-aggregates-and-pods-and-how-why-are-they-special>

⁶<https://godbolt.org/z/WW569j6h6>

It's because `{40, '*'}` converts 40 into a character (using its ASCII code) and passes those two characters through `std::initializer_list` to create a string with two characters only. The problem is that direct initialization with parens (parentheses) won't work inside a class member declaration:

```
class DataPacket {
    std::string data_ (40, '*'); // syntax error!

/* rest of the code*/
```

The code doesn't compile and to fix this you can rely on copy initialization:

```
class DataPacket {
    std::string data_ = std::string(40, '*'); // fine

/* rest of the code*/
```

This limitation might be related to the fact that the syntax parens might quickly run into the most vexing parse/parsing issues, which might be even worse for class members.

72.3 3. No deduction for NSDMI

You can use `auto` for static variables:

```
class Type {
    static inline auto theMeaningOfLife = 42; // int deduced
};
```

However, you cannot use it as a class non-static member:

```
class Type {
    auto myField { 0 }; // error
    auto param { 10.5f }; // error
};
```

The alternative syntax also fails:

```
class Type {
    auto myField = int { 10 };
};
```

Similarly for CTAD (from C++17). it works fine for `static` data members of a class:

```
class Type {
    static inline std::vector ints { 1, 2, 3, 4, 5 }; // deduced vector<int>
};
```

However, it does not work as a non-static member:

```
class Type {
    std::vector ints { 1, 2, 3, 4, 5 }; // syntax error!
};
```

Same happens for arrays, the compiler cannot deduce the number of elements nor the type:

```
struct Wrapper {
    int numbers[] = {1, 2, 3, 4}; // syntax error!
    std::array nums { 0.1f, 0.2f, 0.3f }; // error...
};
```

72.4 4. List initialization. Is it uniform?

Since C++11, we have a new way of initialization, called list initialization {}. Sometimes called brace initialization or even uniform initialization.

Is it really uniform?

In most places, you can use it...and with each C++ standard, the rules are less confusing...unless you have an exception.

For example:

```
1 int x0 { 78.5f }; // error, narrowing conversion
2 auto x1 = { 1, 2 }; // decltype(x1) is std::initializer_list<int>
3 auto x2 = { 1, 2.0 }; // error: cannot deduce element type
4 auto x3{ 1, 2 }; // error: not a single element (since C++17)
5 auto x4 = { 3 }; // decltype(x4) is std::initializer_list<int>
6 auto x5{ 3 }; // decltype(x5) is int (since C++17)
```

Additionally there's this famous issue with a vector:

```
std::vector<int> vec1 { 1, 2 }; // holds two values, 1 and 2
std::vector<int> vec2 ( 1, 2 ); // holds one value, 2
```

For data members, there's no `auto` type deduction nor CTAD, so we have to specify the exact type of a member. I think list initialization is more uniform and less problematic in this case.

Some summary:

- Initialization in C++ is bonkers⁷ - a famous article where it listed eighteen different forms of initialization (as of C++14).
- In Item 7 for Effective Modern C++, Scott Meyers said that “braced initialization is the most widely usable initialization syntax, it prevents narrowing conversions, and it’s immune to C++’s most vexing parse.
- Nicolai Josuttis had an excellent presentation about all corner cases: CppCon 2018: Nicolai Josuttis “The Nightmare of Initialization in C++” - YouTube⁸, and suggests using
- Core Guidelines: C++ Core Guidelines - ES.23: Prefer the {}-initializer syntax⁹. Exception: For containers, there is a tradition for using {...} for a list of elements and (...) for sizes.

⁷<https://blog.tartanllama.xyzinitialization-is-bonkers/>

⁸<https://www.youtube.com/watch?v=7DTIWPGX6zs>

⁹<https://isocpp.github.ioCppCoreGuidelines/CppCoreGuidelines#es23-prefer-the-initializer-syntax>

- Initialization of a variable declared using `auto` with a single value, e.g., `{v}`, had surprising results until C++17. The C++17 rules are somewhat less surprising.
- Only abseil / Tip of the Week #88: Initialization: `=`, `()`, and ¹⁰ - prefers the old style. This guideline was updated in 2015, so many things were updated as of C++17 and C++20.
- In Core C++ 2019 :: Timur Doumler :: Initialisation in modern C++ - YouTube¹¹ - Timur suggests for all, but if you want to be sure about the constructor being called then use `()`. As `()` performs regular overload resolution.

In the book about data members, I follow the rule to use `{}` in most places unless it's obvious to use `()` to call some proper constructor.

72.5 5. `std::initializer_list` is greedy

All containers from the Standard Library have constructors supporting `initializer_list`. For instance:

```

1 // the vector class:
2 constexpr vector( std::initializer_list<T> init,
3                     const Allocator& alloc = Allocator() );
4
5 // map:
6 map( std::initializer_list<value_type> init,
7         const Compare& comp = Compare(),
8         const Allocator& alloc = Allocator() );
```

We can create our own class and simulate this behaviour:

```

1 #include <iostream>
2 #include <initializer_list>
3
4 struct X {
5     X(std::initializer_list<int> list)
6     : count{list.size()} { puts("X(init_list)"); }
7     X(size_t cnt) : count{cnt} { puts("X(cnt)"); }
8     X() { puts("X()"); }
9     size_t count {};
10 };
11
12 int main() {
13     X x;
14     std::cout << "x.count = " << x.count << '\n';
15     X y { 1 };
```

¹⁰<https://abseil.io/tips/88>

¹¹<https://www.youtube.com/watch?v=v0jM4wm1zYA>

```

16     std::cout << "y.count = " << y.count << '\n';
17     X z { 1, 2, 3, 4 };
18     std::cout << "z.count = " << z.count << '\n';
19     X w ( 3 );
20     std::cout << "w.count = " << w.count << '\n';
21 }
```

Run @Compiler Explorer¹²

The X class defines three constructors, and one of them takes `initializer_list`. If we run the program, you'll see the following output:

```

1 X()
2 x.count = 0
3 X(initializer_list)
4 y.count = 1
5 X(initializer_list)
6 z.count = 4
7 X(cnt)
8 w.count = 3
```

As you can see, writing `X x;` invokes a default constructor. Similarly, if you write `X x{};`, the compiler won't call a constructor with the empty initializer list. But in other cases, the list constructor is “greedy” and will take precedence over the regular constructor taking one argument. To call the exact constructor, you need to use direct initialization with parens `()`.

72.6 Summary

In the article, we touched on important topics like aggregates, non-static data member initialization, and a few others. This is definitely not all; for example, C++20 allows using parentheses lists `(. . .)` to initialize aggregates, and C++17 added `inline` variables.

- Do you use in-class member initialization?
- Have you got any tricks for handling data members?

¹²<https://godbolt.org/z/h1W88Pebq>

Chapter 73

On finding the average of two unsigned integers without overflow

• Raymond Chen  2023-02-07  ★★

Finding the average of two unsigned integers, rounding toward zero, sounds easy:

```
unsigned average(unsigned a, unsigned b)
{
    return (a + b) / 2;
}
```

However, this gives the wrong answer in the face of integer overflow¹: For example, if unsigned integers are 32 bits wide, then it says that `average(0x80000000U, 0x80000000U)` is zero.

If you know which number is the larger number (which is often the case), then you can calculate the width and halve it²:

```
unsigned average(unsigned low, unsigned high)
{
    return low + (high - low) / 2;
}
```

There's another algorithm that doesn't depend on knowing which value is larger, the U.S. patent for which expired in 2016³:

```
unsigned average(unsigned a, unsigned b)
{
    return (a / 2) + (b / 2) + (a & b & 1);
}
```

The trick here is to pre-divide the values before adding. This will be too low if the original addition contained a carry from bit 0 to bit 1, which happens if bit 0 is set in both of the terms, so we detect that case and make the necessary adjustment.

¹<https://devblogs.microsoft.com/oldnewthing/20030917-00/?p=42453>

²<https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html>

³<https://patents.google.com/patent/US6007232A/en>

And then there's the technique in the style known as SWAR⁴, which stands for “SIMD within a register”.

```
unsigned average(unsigned a, unsigned b)
{
    return (a & b) + (a ^ b) / 2;
}
```

If your compiler supports integers larger than the size of an `unsigned`, say because `unsigned` is a 32-bit value but the native register size is 64-bit, or because the compiler supports multiword arithmetic, then you can cast to the larger data type:

```
unsigned average(unsigned a, unsigned b)
{
    // Suppose "unsigned" is a 32-bit type and
    // "unsigned long long" is a 64-bit type.
    return ((unsigned long long)a + b) / 2;
}
```

The results would look something like this for processor with native 64-bit registers. (I follow the processor's natural calling convention for what is in the upper 32 bits of 64-bit registers.)

```
1 // x86-64: Assume ecx = a, edx = b, upper 32 bits unknown
2     mov    eax, ecx      ; rax = ecx zero-extended to 64-bit value
3     mov    edx, edx      ; rdx = edx zero-extended to 64-bit value
4     add    rax, rdx      ; 64-bit addition: rax = rax + rdx
5     shr    rax, 1        ; 64-bit shift:    rax = rax >> 1
6                           ; result is zero-extended
7                           ; Answer in eax
8
9 // AArch64 (ARM 64-bit): Assume w0 = a, w1 = b, upper 32 bits unknown
10    uxtw  x0, w0        ; x0 = w0 zero-extended to 64-bit value
11    add   x0, w1, uxtw   ; 64-bit addition: x0 = x0 + (uint32_t)w1
12    ubfx  x0, x0, 1, 32  ; Extract bits 1 through 32 from result
                           ; (shift + zero-extend in one instruction)
13                           ; Answer in x0
14
15
16 // Alpha AXP: Assume a0 = a, a1 = b, both in canonical form
17    insll a0, #0, a0      ; a0 = a0 zero-extended to 64-bit value
18    insll a1, #0, a1      ; a1 = a1 zero-extended to 64-bit value
19    addq  a0, a1, v0      ; 64-bit addition: v0 = a0 + a1
20    srl   v0, #1, v0      ; 64-bit shift:    v0 = v0 >> 1
21    addl  zero, v0, v0    ; Force canonical form
                           ; Answer in v0
```

⁴<https://en.wikipedia.org/wiki/SWAR>

```

23
24 // MIPS64: Assume a0 = a, a1 = b, sign-extended
25   dext  a0, a0, 0, 32 ; Zero-extend a0 to 64-bit value
26   dext  a1, a1, 0, 32 ; Zero-extend a1 to 64-bit value
27   addu v0, a0, a1      ; 64-bit addition: v0 = a0 + a1
28   dsrl v0, v0, #1      ; 64-bit shift:    v0 = v0 >> 1
29   sll   v0, #0, v0      ; Sign-extend result
30                           ; Answer in v0
31
32 // Power64: Assume r3 = a, r4 = b, zero-extended
33   add   r3, r3, r4      ; 64-bit addition: r3 = r3 + r4
34   rldicl r3, r3, 63, 32 ; Extract bits 63 through 32 from result
35                           ; (shift + zero-extend in one instruction)
36                           ; result in r3
37
38 // Itanium Ia64: Assume r32 = a, r4 = b, upper 32 bits unknown
39   extr r32 = r32, 0, 32  ; zero-extend r32 to 64-bit value
40   extr r33 = r33, 0, 32  ; zero-extend r33 to 64-bit value
41   add.i8 r8 = r32, r33 ; ; 64-bit addition: r8 = r32 + r33
42   shr   r8 = r8, 1      ; 64-bit shift:    r8 = r8 >> 1

```

Note that we must ensure that the upper 32 bits of the 64-bit registers are zero, so that any leftover values in bit 32 don't infect the sum. The instructions to zero out the upper 32 bits may be elided if you know ahead of time that they are already zero. This is common on x86-64 and AArch64 since those architectures naturally zero-extend 32-bit values to 64-bit values, but not common on Alpha AXP and MIPS64 because those architectures naturally sign-extend 32-bit values to 64-bit values.

I find it amusing that the PowerPC, patron saint⁵ of ridiculous instructions⁶, has an instruction whose name almost literally proclaims its ridiculousness: rldicl. (It stands for “rotate left doubleword by immediate and clear left”.)

For 32-bit processors with compiler support for multiword arithmetic, you end up with something like this:

```

1 // x86-32
2   mov   eax, a          ; eax = a
3   xor   ecx, ecx        ; Zero-extend to 64 bits
4   add   eax, b          ; Accumulate low 32 bits in eax, set carry on overflow
5   adc   ecx, ecx        ; Accumulate high 32 bits in ecx
6                           ; ecx:eax = 64-bit result
7   shrd  eax, ecx, 1     ; Multiword shift right
8                           ; Answer in eax
9
10 // ARM 32-bit: Assume r0 = a, r1 = b

```

⁵<https://devblogs.microsoft.com/oldnewthing/20180815-00/?p=99495>

⁶<https://devblogs.microsoft.com/oldnewthing/20180814-00/?p=99485>

```

11      mov     r2, #0          ; r2 = 0
12      adds   r0, r1, r2      ; Accumulate low 32 bits in r0, set carry on overflow
13      adc    r1, r2, #0      ; Accumulate high 32 bits in r1
14                  ; r1:r0 = 64-bit result
15      lsrs   r1, r1, #1      ; Shift high 32 bits right one position
16                  ; Bottom bit goes into carry
17      rrx    r0, r0          ; Rotate bottom 32 bits right one position
18                  ; Carry bit goes into top bit
19                  ; Answer in r0
20
21 // SH-3: Assume r4 = a, r5 = b
22 ; (MSVC 13.10.3343 code generation here isn't that great)
23      clrt              ; Clear T flag
24      mov    #0, r3          ; r3 = 0, zero-extended high 32 bits of a
25      addc  r5, r4          ; r4 = r4 + r5 + T, overflow goes into T bit
26      mov    #0, r2          ; r2 = 0, zero-extended high 32 bits of b
27      addc  r3, r2          ; r2 = r2 + r3 + T, calculate high 32 bits
28                  ; r3:r2 = 64-bit result
29      mov    #31, r3          ; Prepare for left shift
30      shld   r3, r2          ; r2 = r2 << r3
31      shlr   r4              ; r4 = r4 >> 1
32      mov    r2, r0          ; r0 = r2
33      or     r4, r0          ; r0 = r0 | r4
34                  ; Answer in r0
35
36 // MIPS: Assume a0 = a, a1 = b
37      addu  v0, a0, a1      ; v0 = a0 + a1
38      sltu  a0, v0, a0      ; a0 = 1 if overflow occurred
39      sll   a0, 31          ; Move to bit 31
40      srl   v0, v0, #1      ; Shift low 32 bits right one position
41      or    v0, v0, a0      ; Combine the two parts
42                  ; Answer in v0
43
44 // PowerPC: Assume r3 = a, r4 = b
45 ; (gcc 4.8.5 -O3 code generation here isn't that great)
46      mr    r9, r3          ; r9 = r3 (low 32 bits of 64-bit a)
47      mr    r11, r4          ; r11 = r4 (low 32 bits of 64-bit b)
48      li    r8, #0          ; r8 = 0 (high 32 bits of 64-bit a)
49      li    r10, #0         ; r10 = 0 (high 32 bits of 64-bit b)
50      addc r11, r11, r9      ; r11 = r11 + r9, set carry on overflow
51      adde r10, r10, r8      ; r10 = r10 + r8, high 32 bits of 64-bit result
52      rlwinm r3, r10, 31, 1, 31 ; r3 = r10 >> 1

```

```

53    rlwinm  r9, r11, 31, 0, 0 ; r9 = r1 << 31
54    or      r3, r3, r9       ; Combine the two parts
55                                ; Answer in r3
56
57 // RISC-V: Assume a0 = a, a1 = b
58    add    a1, a0, a1       ; a1 = a0 + a1
59    sltu  a0, a1, a0       ; a0 = 1 if overflow occurred
60    slli   a0, a0, 31       ; Shift to bit 31
61    srli   a1, a1, 1        ; a1 = a1 >> 1
62    or     a0, a0, a1       ; Combine the two parts
63                                ; Answer in a0

```

Or if you have access to SIMD registers that are larger than the native register size, you can do the math there. (Though crossing the boundary from general-purpose register to SIMD register and back may end up too costly.)

```

// x86-32
unsigned average(unsigned a, unsigned b)
{
    auto a128 = _mm_cvtsi32_si128(a);
    auto b128 = _mm_cvtsi32_si128(b);
    auto sum = _mm_add_epi64(a128, b128);
    auto avg = _mm_srli_epi64(sum, 1);
    return _mm_cvtsi128_si32(avg);
}

1    movd   xmm0, a          ; Load a into bottom 32 bits of 128-bit register
2    movd   xmm1, b          ; Load b into bottom 32 bits of 128-bit register
3    paddq  xmm1, xmm0       ; Add as 64-bit integers
4    psrlq  xmm1, 1          ; Shift 64-bit integer right one position
5    movd   eax, xmm1         ; Extract bottom 32 bits of result

// 32-bit ARM (A32) has an "average" instruction built in
unsigned average(unsigned a, unsigned b)
{
    auto a64 = vdup_n_u32(a);
    auto b64 = vdup_n_u32(b);
    auto avg = vhadd_u32(a64, b64); // hadd = half of add (average)
    return vget_lane_u32(avg);
}

1    vdup.32 d16, r0          ; Broadcast r0 into both halves of d16
2    vdup.32 d17, r1          ; Broadcast r1 into both halves of d17
3    vhadd.u32 d16, d16, d17 ; d16 = average of d16 and d17
4    vmov.32 r0, d16[0]       ; Extract result

```

But you can still do better, if only you had access to better intrinsics.

In processors that support add-with-carry, you can view the sum of register-sized integers as a $(N + 1)$ -bit result, where the bonus bit N is the carry bit. If the processor also supports rotate-right-through-carry, you can shift $(N + 1)$ -bit result right one place, recovering the correct average without losing the bit that overflows.

```

1 // x86-32
2     mov    eax, a
3     add    eax, b          ; Add, overflow goes into carry bit
4     rcr    eax, 1          ; Rotate right one place through carry
5
6 // x86-64
7     mov    rax, a
8     add    rax, b          ; Add, overflow goes into carry bit
9     rcr    rax, 1          ; Rotate right one place through carry
10
11 // 32-bit ARM (A32)
12    mov    r0, a
13    adds  r0, b          ; Add, overflow goes into carry bit
14    rrx   r0              ; Rotate right one place through carry
15
16 // SH-3
17    clrt             ; Clear T flag
18    mov    a, r0
19    addc  b, r0          ; r0 = r0 + b + T, overflow goes into T bit
20    rotcr r0            ; Rotate right one place through carry

```

While there is an intrinsic for the operation of “add two values and report the result as well as carry”, we don’t have one for “rotate right through carry”, so we can get only halfway there:

```

1 unsigned average(unsigned a, unsigned b)
2 {
3 #if defined(_MSC_VER)
4     unsigned sum;
5     auto carry = _addcarry_u32(0, a, b, &sum);
6     return _rotr1_carry(sum, carry); // missing intrinsic!
7 #elif defined(__clang__)
8     unsigned carry;
9     auto sum = __builtin_adc(a, b, 0, &carry);
10    return __builtin_rotateright1throughcarry(sum, carry); // missing intrinsic!
11 #elif defined(__GNUC__)
12     unsigned sum;
13     auto carry = __builtin_add_overflow(a, b, &sum);
14     return __builtin_rotateright1throughcarry(sum, carry); // missing intrinsic!
15 #else

```

```

16 #error Unsupported compiler.
17 #endif
18 }

```

We'll have to fake it, alas. Here's one way:

```

1 unsigned average(unsigned a, unsigned b)
2 {
3 #if defined(_MSC_VER)
4     unsigned sum;
5     auto carry = _addcarry_u32(0, a, b, &sum);
6     return (sum / 2) | (carry << 31);
7 #elif defined(__clang__)
8     unsigned carry;
9     auto sum = __builtin_addc(a, b, 0, &carry);
10    return (sum / 2) | (carry << 31);
11 #elif defined(__GNUC__)
12     unsigned sum;
13     auto carry = __builtin_add_overflow(a, b, &sum);
14     return (sum / 2) | (carry << 31);
15 #else
16 #error Unsupported compiler.
17 #endif
18 }

// _MSC_VER
1     mov     ecx, a
2     add     ecx, b          ; Add, overflow goes into carry bit
3     setc    al              ; al = 1 if carry set
4     shr     ecx, 1          ; Shift sum right one position
5     movzx   eax, al          ; eax = 1 if carry set
6     shl     eax, 31          ; Move to bit 31
7     or      eax, ecx         ; Combine
8                               ; Result in eax

// __clang__
1     mov     ecx, a
2     add     ecx, b          ; Add, overflow goes into carry bit
3     setc    al              ; al = 1 if carry set
4     shld   eax, ecx, 31      ; Shift left 64-bit value
5                               ; Result in eax

// __clang__ with ARM-Thumb2
1     adds   r0, r0, r1        ; Calculate sum with flags
2     blo    nope             ; Jump if carry clear

```

```

4     movs    r1, #1          ; Carry is 1
5     lsls    r1, r1, #31      ; Move carry to bit 31
6     lsrs    r0, r0, #1          ; Shift sum right one position
7     adcs    r0, r0, r1      ; Combine
8     b      done

9  nope:
10    movs   r1, #0          ; Carry is 0
11    lsrs   r0, r0, #1          ; Shift sum right one position
12    adds   r0, r0, r1      ; Combine

13 done:

14

15 // __GNUC__
16    mov    eax, a
17    xor    edx, edx      ; Preset edx = 0 for later setc
18    add    eax, b          ; Add, overflow goes into carry bit
19    setc   dl           ; dl = 1 if carry set
20    shr    eax, 1          ; Shift sum right one position
21    shl    edx, 31      ; Move carry to bit 31
22    or     eax, edx      ; Combine

```

I considered trying a sneaky trick: Use the rotation intrinsic. (gcc doesn't have a rotation intrinsic, so I couldn't try it there.)

```

1  unsigned average(unsigned a, unsigned b)
2  {
3  #if defined(_MSC_VER)
4      unsigned sum;
5      auto carry = _addcarry_u32(0, a, b, &sum);
6      sum = (sum & ~1) | carry;
7      return _rotr(sum, 1);
8  #elif defined(__clang__)
9      unsigned carry;
10     sum = (sum & ~1) | carry;
11     auto sum = __builtin_addc(a, b, 0, &carry);
12     return __builtin_rotateright32(sum, 1);
13 #else
14 #error Unsupported compiler.
15 #endif
16 }

1 // _MSC_VER
2     mov    ecx, a
3     add    ecx, b          ; Add, overflow goes into carry bit
4     setc   al           ; al = 1 if carry set

```

```

5      and    ecx, -2          ; Clear bottom bit
6      movzx  ecx, al         ; Zero-extend byte to 32-bit value
7      or     eax, ecx        ; Combine
8      ror    ear, 1          ; Rotate right one position
9                      ; Result in eax

1 // __clang__
2      mov    ecx, a
3      add    ecx, b          ; Add, overflow goes into carry bit
4      setc   al              ; al = 1 if carry set
5      shld  eax, ecx, 31     ; Shift left 64-bit value

1 // __clang__ with ARM-Thumb2
2      movs   r2, #0           ; Prepare to receive carry
3      adds   r0, r0, r1        ; Calculate sum with flags
4      adcs   r2, r2           ; r2 holds carry
5      lsrs   r0, r0, #1        ; Shift sum right one position
6      lsls   r1, r2, #31        ; Move carry to bit 31
7      adds   r0, r1, r0        ; Combine

```

Mixed results. For `_MSC_VER`, the code generation got worse. For `__clang__` for ARM-Thumb2, the code generation got better. And for `__clang__` for x86, the compiler realized that it was the same as before, so it just used the previous codegen!

Bonus chatter: And while I’m here, here are sequences for processors that don’t have rotate-right-through-carry.

```

1 // AArch64 (A64)
2      mov    x0, a
3      adds  x0, x1, b          ; Add, overflow goes into carry bit
4      addc  x1, xzr, xzr        ; Copy carry to x1
5      extr  x0, x1, x0, 1        ; Extract bits 64:1 from x1:x0
6                      ; Answer in x0

1 // Alpha AXP: Assume a0 = a, a1 = b, both 64-bit values
2      addq   a0, a1, v0          ; 64-bit addition: v0 = a0 + a1
3      cmpult a0, v0, a0          ; a0 = 1 if overflow occurred
4      srl    v0, #1, v0           ; 64-bit shift: v0 = v0 >> 1
5      sll    a0, #63, a0          ; 64-bit shift: a0 = a0 << 63
6      or     a0, v0, v0           ; v0 = v0 | a0
7                      ; Answer in v0

1 // Itanium Ia64: Assume r32 = a, r33 = b, both 64-bit values
2      add    r8 = r32, r33 ;;;    ; 64-bit addition: r8 = r32 + r33
3      cmp.ltu p6, p7 = r8, r33 ;;; // p6 = true if overflow occurred
4      (p6) addl  r9 = 1, r0          ; r9 = 1 if overflow occurred
5      (p7) addl  r9 = 0, r0 ;;;      ; r9 = 0 if overflow did not occur

```

```

6     shrp    r8 = r9, r8, 1      // r8 = extract bits 64:1 from r9:r8
7                           // Answer in r8

1 // MIPS: Same as multiprecision version

1 // PowerPC: Assume r3 = a, r4 = b
2     addc   r3, r3, r4          ; Accumulate low 32 bits in r3, set carry on overflow
3     adde   r5, r4, r4          ; Shift carry into bottom bit of r5 (other bits garbage)
4     rlwinm r3, r3, 31, 1, 31   ; Shift r3 right by one position
5     rlwinm r5, r5, 31, 0, 0    ; Shift bottom bit of r5 to bit 31
6     or     r3, r5, r5          ; Combine the two parts

1 // RISC-V: Same as multiprecision version

```

Bonus chatter: C++20 adds a `std::midpoint` function that calculates the average of two values (rounding toward a).

Bonus viewing: `std::midpoint`? How hard could it be?⁷

Update: I was able to trim an instruction off the PowerPC version by realizing that only the bottom bit of `r5` participates in the `rlwinm`, so the other bits can be uninitialized garbage. For the uninitialized garbage, I used `r4`, which I know can be consumed without a stall because the `addc` already consumed it.

Here's the original:

```

1 // PowerPC: Assume r3 = a, r4 = b
2     li     r5, #0             ; r5 = 0 (accumulates high 32 bits)
3     addc   r3, r3, r4          ; Accumulate low 32 bits in r3, set carry on overflow
4     addze  r5, r5              ; Accumulate high bits in r5
5     rlwinm r3, r3, 31, 1, 31   ; Shift r3 right by one position
6     rlwinm r5, r5, 31, 0, 0    ; Shift bottom bit of r5 to bit 31
7     or     r3, r5, r5          ; Combine the two parts

```

Update 2: Peter Cordes pointed out that an instruction can also be trimmed from the AArch64 version by using the `uxtw` extended register operation to combine a `uxtw` with an `add`. Here's the original:

```

1 // AArch64 (ARM 64-bit): Assume w0 = a, w1 = b, upper 32 bits unknown
2     uxtw   x0, w0              ; x0 = w0 zero-extended to 64-bit value
3     uxtw   x1, w1              ; x1 = w1 zero-extended to 64-bit value
4     add    x0, x1              ; 64-bit addition: x0 = x0 + x1
5     ubfx   x0, x0, 1, 32        ; Extract bits 1 through 32 from result
6                               ; (shift + zero-extend in one instruction)
7                               ; Answer in x0

```

⁷<https://www.youtube.com/watch?v=sBtAGxBh-XI>

Part X

Miscellaneous

一曲将尽，曲谐终章，为免虎头蛇尾，本 Part 精挑细选了无法归类到其他主题的优秀文章。

你可于此比较各种最近的 C++ Successor 新语言，了解旧组件的缺陷和新组件的解决之法，学习编译器的编译标志，领略复杂的编码世界，知悉 C 的最新动向……

内容亦是包罗万象，各章相互独立，因此建议先从目录着手，挑取感兴趣的文章阅读。其中有些文章看似内容庞杂，但繁杂的背后是类库设计者在性能、安全、易用、通用、可移植性之间做出的艰难权衡取舍；有些文章内容虽浅显易懂，但也需多年经验，才能识其妙处所在；有些文章虽无关 C++，但却反应了很多 C++ 资深使用者的反思和探索。

以此文章，飨以读者，望袅袅余音，伴随的是各位的满载而归。

2023 年 5 月 14 日
邹启翔

Chapter 74

New integer types I'd like to see

👤 Jonathan 📅 2022-09-29 💬 ★★

(Most) C++ implementations provide at least 8, 16, 32, and 64-bit signed and unsigned integer types. There are annoying implicit conversions, discussions about undefined behavior on overflow (some think it's too much UB, others think it's not enough), but for the most part they do the job well. Newer languages like Rust copied that design, but fixed the conversions and overflow behavior.

Still, I think there is room for innovation here. Let me talk about three new families of integer types I'd like to see.

74.1 Symmetric signed integers

The de-facto representation of signed integers on modern hardware is two's complement. There positive values have the most significant bit set to zero, while it is set for negative values. To get the absolute value of a negative number, flip all bits and add one.

For example, on an 8-bit integer, 42 is 0b0'0101010: sign bit zero, the rest representing 42 in binary. On the other hand, -42 is 0b1'1010110: if you flip all bits you get 0b0'0101001, add one and you're back at 0b0'0101010, which is 42. Crucially, 0b1'1111111 is -1 and more negative values go down to 0b1'0000000, which is -128.

Notice something interesting about the last value? If you do the conversion, flipping gives you 0b0'1111111, and the addition of one results in 0b1'0000000 –overflowing into the sign bit!

This means there absolute value of the smallest value is bigger than the absolute value of the biggest value 0b0'1111111 or 127; there are more negative values than positive values, because the positive half where the sign bit isn't set also contain the number zero.

I really don't like this asymmetry –it leads to annoying edge cases in all sort of integer APIs.

For starters, `abs(x)` for integers `x` isn't a total function: `abs(INT_MIN)` isn't representable. Likewise, `x * (-1)` isn't total either: `INT_MIN * (-1)` overflows. It gets even funnier when you consider division: surely `x / y` cannot overflow as it division makes things smaller, right? Wrong, `INT_MIN / (-1)` overflows (and raises a division by zero (!) error on x86). Furthermore, `INT_MIN % (-1)` is also UB.

So here's my wish: a signed integer where `INT_MIN == -INT_MAX`, by moving the minimal value one higher.

First, you’re not losing anything useful: I’d argue the extra negative number isn’t important in most use cases. After all, if you had an extra number, wouldn’t it make more sense to have an additional positive number instead of a negative number?

Second, you’re getting symmetry back. All the operations mentioned above are now symmetric and can’t overflow. This makes them a lot easier to reason about.

Third, you’re getting an unused bit pattern `0b1'0000000`, the old `INT_MIN`, which you can interpret however you like. While you in principle could turn it into some sort of negative zero for extra symmetry, please don’t (just use one’s complement or sign magnitude instead). Instead we should copy a different feature from floating point arithmetic: not-a-number or `Nan`. Let’s call it `INT_NAN = 0b1'0000000`.

Just like floating point’s `Nan`, `INT_NAN` isn’t a valid integer value. In an ideal world, it would also be sticky on arithmetic, so `INT_NAN ⊞ x == INT_NAN`, but that requires hardware support for efficiency. Instead, let’s just say arithmetic on `INT_NAN` is undefined behavior; sanitizers can then insert assertions in debug mode.

Why do I want `INT_NAN` and thus add an additional precondition to every integer arithmetic?

Because I really like sentinel values.

For example, with `INT_NAN`, it is possible to have `sizeof(std::optional<int>) == int`: instead of having to store an additional boolean to keep track of the existence of an optional, we can just store `INT_NAN` instead. Likewise, a closed hash table needs some way to distinguish between empty and non-empty entries. Having a sentinel value removes the need for additional meta data.

Now you might not like it that we’re picking a sentinel value here. What if you want to store `INT_NAN` in an `std::optional<int>`?

Well, `INT_NAN` isn’t a number, so why do you want to store it in an `int`? Only if you need some sort of sentinel value on your own. This is similar to `Nan` boxing¹ of floating point values –you lose the ability to store (most) `Nan`s, but gain more efficient storage. However, unlike floating point arithmetic where e.g. `0/0` can result in `Nan`, under my model, no arithmetic operation on integers can result in `INT_NAN` as overflow is undefined behavior. So you really need to get out of your way by assigning `INT_NAN` to introduce integer `Nan`’s in your code.

You might not like that I suggest arithmetic on `INT_NAN` should be UB if you’ve been burned by aggressive compiler optimizations in the past. However, UB in the standard by itself is not a bad thing; UB literally means that the standard poses no requirements on the behavior, which gives the compiler the most freedom. They can assume it does not happen and optimize accordingly, but they can also insert debug assertions (either catching all, or rough checks with false negatives), or give it well-defined behavior. Most mainstream compilers do the first interpretation by default, but for example *I’m currently working on a C interpreter*², where it will panic on all instances on UB.

74.2 Unsigned integers with one bit missing

Using unsigned integer types in C++ is controversial, to say the least. An argument in favor is the ability to express in the type system in the type system when something cannot be negative. An argument against

¹<https://anniecherkaev.com/the-secret-life-of-nan>

²https://www.youtube.com/watch?v=sCDsMc61iWM&list=PLbxut1xyrkCZ-9d_03G0KBU4uh782J1eN

is the fact that subtraction easily results in big values due to integer overflow at zero.

Unrelated, but “integer underflow” is not a thing. Exceeding the minimal value of an integer is still integer overflow. Underflow³ occurs when you have a number that is too close to zero to be represented as a floating point.

As a proponent of unsigned integer types, I can’t argue against the annoying overflow on subtraction. It can cause all sorts of nasty bugs from buffer overflow to out of memory errors. Switching to a signed integer makes sense here as a very negative value makes it really obvious that an error occurred, and that is also the position many people take and use signed for everything. This is a shame, since you lose the ability to express yourself in the type system.

Since you apparently don’t need the extra bit of storage space in many situations, I’d like to have something that is logically a 63 bit unsigned integers as opposed to a 64 bit one. In assembly, it is represented the same way a 64 bit signed integer would. However, it is undefined behavior if it ever stores a negative value. This is similar to `bool`: it is logically a single bit but physically represented as a byte.

This sounds just like a signed integer with extra steps and more UB, so why bother?

First, compared to `int`, it automatically comes with a precondition that it cannot be negative. Second, fewer overflow checks are necessary in debug mode. Since we’re having a wide range of invalid values, we can just check the value whenever we do a store operation and not after every single arithmetic operation. Sure, we could overflow in an intermediate expression and then undo the overflow in subsequent arithmetic, but it is a nice trade-off between performance and safety.

Again, this sort of lax debug check is not possible if the standard were to require program termination on overflow.

In fact, those unsigned integer types are exactly equivalent to using the corresponding signed integer type and asserting that it is non-negative whenever necessary. It is just built into the type system and implemented by the compiler, instead of a programmer written precondition.

74.3 Distinct bit vectors vs integer type

I’ve recently worked on a compiler for a language that makes a distinction between signed integer types, unsigned integer types, and bit integer types. The first two families support only arithmetic operations while only the last one support bit operations. I was skeptical at first but came to really like the distinction.

I always found it weird how we treat an integer type both as a number and do arithmetic on it while also modifying individual bits. When you’re doing math, you rarely need to modify individual digits! This is especially true with signed integers, where the sign bit messes everything up and leads to implementation-defined or undefined behavior on shift operations.

Sure, distinguishing the two makes writing optimizations like shift instead of division or bitwise and instead of modulo more annoying to write, and some fancy bit hacks require more casts, but it also makes it really obvious what’s going on: you’re starting to treat an integer as a sequence of bits for some performance benefit, which requires some sort of documentation.

³https://en.wikipedia.org/wiki/Arithmetic_underflow

When I mentioned the shift optimizations, I’m not talking about replacing `x / 2` by `x >> 1` – the compiler is going to do that. I’m talking about things like `hash % hash_table_size`, where `hash_table_size` is always a power of two, but the compiler can’t know that.

While we’re at it: why do we reserve so many tokens for bit operations? How often do you actually need `|`, `&` or `~` to warrant an entire character that can’t be used for anything else? Not to mention that they have the *wrong precedence in C*⁴, and aren’t really useful on their own: you often want `is_bit_set(x, n)`, `extract_bits(x, low, high)` or other higher-level operations built on top of the bitwise operations. I’d like to see the operations delegated to (built-in) standard library functions, so we can re-use the operators for something else like pipelines.

74.4 Conclusion

There are a lot of new languages⁵ popping up in the C++ space recently, I’d love to see some of them experiment in such a fundamental area.

Symmetric signed integers make so many fundamental APIs nicer, and a 63 bit unsigned `size_t` can combine the best of the signed/unsigned world for containers. Sure, you still want the real unsigned types in situations where you want the extra bit since it doubles the range, but I think it would be fine to not have the true `INT_MIN` except for interop with C. Distinct bit vectors can make code more expressive, but the casts can also get really annoying. I’d still like to see someone try.

⁴<https://softwareengineering.stackexchange.com/questions/194635/why-do-bitwise-operators-have-lower-priority-than-comparisons>

⁵carbon,cppfront,val...

Chapter 75

proxy: Runtime Polymorphism Made Easier Than Ever

• Mingxin Wang  2022-8-15  ★★★

proxy is an open-source, cross-platform, single-header C++ library, making runtime polymorphism easier to implement and faster, empowered by our breakthrough innovation of Object-oriented Programming (OOP) theory in recent years. Consider three questions:

1. Do you want to facilitate architecture design and maintenance by writing non-intrusive polymorphic code in C++ as easily as in Rust or Golang?
2. Do you want to facilitate lifetime management of polymorphic objects as easily as in languages with runtime Garbage Collection (GC, like Java or C#), without compromising performance?
3. Have you tried other polymorphic programming libraries in C++ but found them deficient?

If so, this library is for you. You can find the implementation at our GitHub repo¹, integrate with your project using vcpkg (search for proxy), or learn more about the theory and technical specifications from P0957².

75.1 Overview

In C++ today, there are certain architecture and performance limitations in existing mechanisms of polymorphism, specifically, virtual functions (based on inheritance) and various polymorphic wrappers (with value semantics) in the standard. As a result, proxy can largely replace the existing “virtual mechanism” to implement your vision in runtime polymorphism, while having no intrusion on existing code, with even better performance.

All the facilities of the library are defined in namespace `pro`. The 3 major class templates are `dispatch`, `facade` and `proxy`. Here is a demo showing how to use this library to implement runtime polymorphism in a different way from the traditional inheritance-based approach:

¹<https://github.com/microsoft/proxy>

²<https://wg21.link/p0957>

```
1 // Abstraction
2 struct Draw : pro::dispatch<void(std::ostream&)> {
3     template <class T>
4         void operator()(const T& self, std::ostream& out) { self.Draw(out); }
5 };
6 struct Area : pro::dispatch<double()> {
7     template <class T>
8         double operator()(const T& self) { return self.Area(); }
9 };
10 struct DrawableFacade : pro::facade<Draw, Area> {};
11
12 // Implementation (No base class)
13 class Rectangle {
14     public:
15         void Draw(std::ostream& out) const
16             { out << "{Rectangle: width = " << width_ << ", height = " << height_ << "}"; }
17         void SetWidth(double width) { width_ = width; }
18         void SetHeight(double height) { height_ = height; }
19         double Area() const { return width_ * height_; }
20
21     private:
22         double width_;
23         double height_;
24 };
25
26 // Client - Consumer
27 std::string PrintDrawableToString(pro::proxy<DrawableFacade> p) {
28     std::stringstream result;
29     result << "shape = ";
30     p.invoke<Draw>(result); // Polymorphic call
31     result << ", area = " << p.invoke<Area>(); // Polymorphic call
32     return std::move(result).str();
33 }
34
35 // Client - Producer
36 pro::proxy<DrawableFacade> CreateRectangleAsDrawable(int width, int height) {
37     Rectangle rect;
38     rect.SetWidth(width);
39     rect.SetHeight(height);
40     return pro::make_proxy<DrawableFacade>(rect); // No heap allocation is expected
41 }
```

75.2 Configure your project

To get started, set the language level of your compiler to at least C++20 and get the header file (proxy.h). You can also install the library via vcpkg, which is a C++ library management software invented by Microsoft, by searching for “proxy” .

To integrate with CMake, 3 steps are required:

1. Set up the vcpkg manifest by adding “proxy” as a dependency in your `vcpkg.json` file:

```
{
    "name": "<project_name>",
    "version": "0.1.0",
    "dependencies": [
        {
            "name": "proxy"
        }
    ]
}
```

2. Use `find_package` and `target_link_libraries` commands to reference to the library proxy in your `CMake-Lists.txt` file:

```
find_package(proxy CONFIG REQUIRED)
target_link_libraries(<target_name> PRIVATE msft_proxy)
```

3. Run CMake with vcpkg toolchain file:

```
cmake <source_dir> \
      -B <build_dir> \
      -DCMAKE_TOOLCHAIN_FILE=<vcpkg_dir>/scripts/buildsystems/vcpkg.cmake
```

75.3 What makes the “proxy” so charming

As a polymorphic programming library, proxy has various highlights, including:

1. being non-intrusive
2. allowing lifetime management per object, complementary with smart pointers
3. high-quality code generation
4. supporting flexible composition of abstractions
5. optimized syntax for Customization Point Objects (CPO) and modules
6. supporting general-purpose static reflection
7. supporting expert performance tuning
8. high-quality diagnostics.

In this section, we will briefly introduce each of the highlights listed above with concrete examples.

75.4 Highlight 1: Being non-intrusive

Designing polymorphic types with inheritance usually requires careful architecting. If the design is not thought through enough early on, the components may become overly complex as more and more functionality is added, or extensibility may be insufficient if polymorphic types are coupled too closely. On the other hand, some libraries (including the standard library) may not have proper polymorphic semantics even if they, by definition, satisfy same specific constraints. In such scenarios, users have no alternative but to design and maintain extra middleware themselves to add polymorphism support to existing implementations.

For example, some programming languages provide base types for containers, which makes it easy for library authors to design APIs without binding to a specific data structure at runtime. However, this is not feasible in C++ because most of the standard containers are not required to have a common base type. I do not think this is a design defect of C++, on the contrary, I think it is reasonable not to overdesign for runtime abstraction before knowing the concrete requirements both for the simplicity of the semantics and for runtime performance. With proxy, because it is non-intrusive, if we want to abstract a mapping data structure from indices to strings for localization, we may define the following facade:

```
struct at : pro::dispatch<std::string(int)> {
    template <class T>
    auto operator()(T& self, int key) { return self.at(key); }
};

struct ResourceDictionaryFacade : pro::facade<at> {};
```

It could proxy any potential mapping data structure, including but not limited to `std::map<int, std::string>`, `std::unordered_map<int, std::string>`, `std::vector<std::string>`, etc.

```
// Library
void DoSomethingWithResourceDictionary(pro::proxy<ResourceDictionaryFacade> p) {
    try {
        std::cout << p.invoke(1) << std::endl;
    } catch (const std::out_of_range& e) {
        std::cout << "No such element: " << e.what() << std::endl;
    }
}

// Client
std::map<int, std::string> var1{{1, "Hello"}};
std::vector<std::string> var2{"I", "love", "Proxy", "!"};
DoSomethingWithResourceDictionary(&var1); // Prints "Hello"
DoSomethingWithResourceDictionary(&var2); // Prints "love"
// Prints "No such element: {implementation-defined error message}"
DoSomethingWithResourceDictionary(
    std::make_shared<std::unordered_map<int, std::string>>());
```

Overall, inheritance-based polymorphism has certain limitations in usability. As Sean Parent com-

mented on NDC 2017³: *The requirements of a polymorphic type, by definition, comes from its use, and there are no polymorphic types, only polymorphic use of similar types. Inheritance is the base class of evil.*

75.5 Highlight 2: Evolutionary lifetime management

It is such a pain to manage lifetime of objects in large systems written in C++. Because C++ does not have built-in GC support due to performance considerations, users need to beware of lifetime management of every single object. Although we have smart pointers since C++11 (i.e., `std::unique_ptr` and `std::shared_ptr`), and various 3rd-party fancy pointers like `boost::interprocess::offset_ptr`, they are not always sufficient for polymorphic use with inheritance. By using the proxy complementary with smart pointers, clients could care less about lifetime management as if there is runtime GC, but without compromising performance.

Before using any polymorphic object, the first step is always to create it. In other programming languages like Java or C#, we can new an object at any time and runtime GC will take care of lifetime management when it becomes unreachable, at the cost of performance. But how should we implement it in C++? Consider the `drawable` example in the “Overview” section: given there are 3 `drawable` types in a system: `Rectangle`, `Circle`, and `Point`. Specifically,

- `Rectangles` have width, height, transparency, and area
- `Circles` have radius, transparency, and area
- `Points` do not have any property; its area is always zero

A library function `MakeDrawableFromCommand` shall be defined as a factory function responsible for creating a drawable instance by parsing the command line.

Here is how we usually define the types with inheritance:

```

1 // Abstraction
2 class IDrawable {
3     public:
4         virtual void Draw(std::ostream& out) const = 0;
5         virtual double Area() const = 0;
6         // Don't forget the virtual destructor, otherwise
7         // `delete`ing a pointer of `IDrawable` may result in memory leak!
8         virtual ~IDrawable() {}
9     };
10
11 // Implementation
12 class Rectangle : public IDrawable {
13     public:
14         void Draw(std::ostream& out) const override;
15         void SetWidth(double width);
16         void SetHeight(double height);
17         void SetTransparency(double);
```

³<https://www.youtube.com/watch?v=QGeVXgEVMJg>

```

18     double Area() const override;
19 }
20
21 class Circle : public IDrawable {
22 public:
23     void Draw(std::ostream& out) const override;
24     void SetRadius(double radius);
25     void SetTransparency(double transparency);
26     double Area() const override;
27 };
28
29 class Point : public IDrawable {
30 public:
31     void Draw(std::ostream& out) const override;
32     constexpr double Area() const override { return 0; }
33 };

```

If we use `std::string` to represent the command line, the parameter type of `MakeDrawableFromCommand` could be `const std::string&`, where there should not be much debate. But what should the return type be? `IDrawable*`? `std::unique_ptr<IDrawable>`? Or `std::shared_ptr<IDrawable>`? Specifically,

- If we use `IDrawable*`, the semantics of the return type is ambiguous because it is a raw pointer type and does not indicate the lifetime of the object. For instance, it could be allocated via `operator new`, from a memory pool or even a global object. Clients always need to learn the hidden contract from the author (or even need to learn the implementation details if the author and documentation are not available for consulting) and properly disposing of the object when the related business has finished via `operator delete` or some other way corresponding to how it was allocated.
- If we use `std::unique_ptr<IDrawable>`, it means every single object is allocated individually from the heap, even if the value is potentially immutable or reusable (“flyweight”), which is potentially bad for performance.
- If we use `std::shared_ptr<IDrawable>`, the performance could become better for flyweight objects due to the relatively low cost of copying, but the ownership of the object becomes ambiguous (a.k.a. “ownership hell”), and the thread-safety guarantee of copy-construction and destruction of `std::shared_ptr` may also add to runtime overhead. On the other hand, if we prefer `std::shared_ptr` across the whole system, every polymorphic type is encouraged to inherit `std::enable_shared_from_this`, which may significantly affect the design and maintenance of a large system.

For proxy, with the definition from the “Overview” section, we can simply define the return type as `pro::proxy<DrawableFacade>` without further concern. In the implementation, `pro::proxy<DrawableFacade>` could be instantiated from all kinds of pointers with potentially different lifetime management strategy. For example, `Rectangles` may be created every time when requested from a memory pool, while the value of `Points` could be cached throughout the lifetime of the program:

```

1 pro::proxy<DrawableFacade> MakeDrawableFromCommand(const std::string& s) {
2     std::vector<std::string> parsed = ParseCommand(s);

```

```

3   if (!parsed.empty()) {
4     if (parsed[0u] == "Rectangle") {
5       if (parsed.size() == 3u) {
6         static std::pmr::unsynchronized_pool_resource rectangle_memory_pool;
7         std::pmr::polymorphic_allocator<> alloc{&rectangle_memory_pool};
8         auto deleter = [alloc](Rectangle* ptr) mutable
9           { alloc.delete_object<Rectangle>(ptr); };
10        Rectangle* instance = alloc.new_object<Rectangle>();
11        // Allocated from a memory pool
12        std::unique_ptr<Rectangle, decltype(deleter)> p{instance, deleter};
13        p->SetWidth(std::stod(parsed[1u]));
14        p->SetHeight(std::stod(parsed[2u]));
15        return p; // Implicit conversion happens
16      }
17    } else if (parsed[0u] == "Circle") {
18      if (parsed.size() == 2u) {
19        Circle circle;
20        circle.SetRadius(std::stod(parsed[1u]));
21        return pro::make_proxy<DrawableFacade>(circle); // SBO may apply
22      }
23    } else if (parsed[0u] == "Point") {
24      if (parsed.size() == 1u) {
25        static Point instance; // Global singleton
26        return &instance;
27      }
28    }
29  }
30  throw std::runtime_error{"Invalid command"};
31 }
```

The full implementation of the example above could be found in our integration tests. In this example, there are 3 return statements in different branches and the return types are also different. Lifetime management with inheritance-based polymorphism is error-prone and inflexible, while proxy allows easy customization of any lifetime management strategy, including but not limited to raw pointers and various smart pointers with potentially pooled memory management.

Specifically, Small Buffer Optimization (SBO, a.k.a., SOO, Small Object Optimization) is a common technique to avoid unnecessary memory allocation (see the second return statement). However, for inheritance-based polymorphism, there are few facilities in the standard that support SBO; for other standard polymorphic wrappers, implementations may support SBO, but there is no standard way to configure it so far. For example, if the size of `std::any` is `n`, it is theoretically impossible to store the concrete value whose size is larger than `n` without external storage.

The top secret making proxy both easy-to-use and fast is that it allows lifetime management per object, which had not been addressed in traditional OOP theory (inheritance-based polymorphism) ever before.

If you have tried other polymorphic programming libraries in C++ before, you may or may not find this highlight of lifetime management unique to proxy. Some of these libraries claim to support various lifetime management model, but do not allow per-object customization like proxy does.

Take dyno⁴ as an example. dyno is another non-intrusive polymorphic programming library in C++. Given an “interface” type I, dyno does not allow `dyno::poly<I>` to have a different lifetime management model. By default, `dyno::poly<I>` always allocates from the heap by the time this blog was written (see `t_yopename Storage = dyno::remote_storage`). For example, if we want to take advantage of SBO, it is needed to override the Storage type, i.e., `dyno::poly<I, dyno::sbo_storage<...>`, which is a different type from `dyno::poly<I>`. Therefore, `dyno::poly<I>` could not be used to implement features like `MakeDrawableFromCommand` above, where the optimal lifetime management model of each branch may differ. Whereas proxy does not have a second template parameter. Given a facade type F, `pro::proxy<F>` is compatible with any lifetime management model within the constraints of the facade.

75.6 Highlight 3: High-quality code generation

Not only does proxy allow efficient lifetime management per object, but also it could generate high quality code for every indirect call. Specifically,

1. Invocations from proxy could be properly inlined, except for the virtual dispatch on the client side, similar to the inheritance-based mechanism.
2. Because proxy is based on pointer semantics, the “dereference” operation may happen inside the virtual dispatch, which always generates different instructions from the inheritance-based mechanism.
3. As tested, with “clang 13.0.0 (x86-64)” and “clang 13.0.0 (RISC-V RV64)”, proxy generates one more instruction than the inheritance-based mechanism, while the situation is reversed with “gcc 11.2 (ARM64)”. This may infer that proxy could have similar runtime performance in invocation with the inheritance-based mechanism at least on the 3 processor architectures (x86-64, ARM64, RISC-V RV64).

More details of code generation analysis could be found in P095⁵.

75.7 Highlight 4: Composition of abstractions

To support reuse of declaration of expression sets, like inheritance of virtual base classes, the facade allows combination of different dispatches with `std::tuple`, while duplication is allowed. For example,

```
struct D1;
struct D2;
struct D3;
struct FA : pro::facade<D1, D2, D3> {};
struct FB : pro::facade<D1, std::tuple<D3, D2>> {};
struct FC : pro::facade<
```

⁴<https://github.com/ldionne/dyno>

⁵<https://wg21.link/p095>

```
std::tuple<D1, D2, D3>, D1, std::tuple<D2, D3>
> {};
```

In the sample code above, given `D1`, `D2` and `D3` are well-formed dispatch types, `FA`, `FB` and `FC` are equivalent. This allows “diamond inheritance” of abstraction without

- syntax ambiguity
- coding techniques like “virtual inheritance”
- extra binary size
- runtime overhead

75.8 Highlight 5: Syntax for CPOs and modules

Along with the standardization of Customization Point Objects (CPO) and improved syntax for Non-Type Template Parameters (NTTP), there are two recommended ways to define a “dispatch” type:

The first way is to manually overload `operator()` as demonstrated before. This is useful when a dispatch is intended to be defined in a header file shared with multiple translation units, e.g., in `tests/proxy_invocation_tests.cpp`:

```
template <class T>
struct ForEach : pro::dispatch<void(pro::proxy<CallableFacade<void(T&)>>) {
    template <class U>
    void operator()(U& self, pro::proxy<CallableFacade<void(T&)>>&& func) {
        for (auto& value : self) {
            func.invoke(value);
        }
    }
};
```

The second way is to specify a `constexpr` callable object as the second template parameter. It provides easier syntax if a corresponding CPO is defined before, or the “dispatch” is intended to be defined in a module with lambda expressions, e.g. in `tests/proxy_invocation_tests.cpp`: `struct GetSize : pro::dispatch<std::size_t(), std::ranges::size> {};`.

75.9 Highlight 6: Static reflection

Reflection is an essential requirement in type erasure, and proxy welcomes general-purpose static (compile-time) reflection other than `std::type_info`.

In other languages like C# or Java, users are allowed to acquire detailed metadata of a type-erased type at runtime with simple APIs, but this is not true for `std::function`, `std::any` or inheritance-based polymorphism in C++. Although these reflection facilities add certain runtime overhead to these languages, they do help users write simple code in certain scenarios. In C++, as the reflection TS keeps evolving, there will be more static reflection facilities in the standard with more specific type information deduced at compile-time than `std::type_info`. It becomes possible for general-purpose reflection to become zero-overhead in C++ polymorphism.

As a result, we decided to make proxy support general-purpose static reflection. It's off by default, and theoretically won't impact runtime performance other than the target binary size if turned on. Here is an example to reflect the given types to `MyReflectionInfo`:

```
class MyReflectionInfo {
public:
    template <class P>
    constexpr explicit MyReflectionInfo(std::in_place_type_t<P>) : type_(typeid(P)) {}
    const char* GetName() const noexcept { return type_.name(); }

private:
    const std::type_info& type_;
};

struct MyFacade : pro::facade</* Omitted */> {
    using reflection_type = MyReflectionInfo;
};
```

Users may call `MyReflectionInfo::GetName()` to get the implementation-defined name of a type at run-time:

```
pro::proxy<MyFacade> p;
puts(p.reflect().GetName()); // Prints typeid(THE_UNDERLYING_POINTER_TYPE).name()
```

75.10 Highlight 7: Performance tuning

To allow implementation balance between extensibility and performance, a set of constraints to a pointer is introduced, including maximum size, maximum alignment, minimum copyability, minimum relocatability and minimum destructibility. The term “relocatability” was introduced in P1144⁶, “equivalent to a move and a destroy”. This blog uses the term “relocatability” but does not depend on the technical specifications of P1144.

While the size and alignment could be described with `std::size_t`, the constraint level of copyability, relocatability and destructibility are described with enum `pro::constraint_level`, which includes `none`, `non-trivial`, `nothrow` and `trivial`, matching the standard wording. The defaults are listed below:

Constraints	Defaults
Maximum size	The size of two pointers
Maximum alignment	The alignment of a pointer
Minimum copyability	None
Minimum relocatability	Nothrow
Minimum destructibility	Nothrow

We can assume the default maximum size and maximum alignment greater than or equal to the implementation of raw pointers, `std::unique_ptr` with default deleters, `std::unique_ptr` with any one-pointer-size of deleters and `std::shared_ptr` of any type.

⁶<https://wg21.link/p1144>

Note that the default minimum copyability is “None”, which means proxy could be instantiated from a non-copyable type like `std::unique_ptr`. However, if we never want to instantiate a proxy with non-copyable types (including `std::unique_ptr`) and want the proxy to be copyable, it is allowed to customize it in a facade definition:

```
// Abstraction
struct MyFacade : pro::facade</* Omitted */> {
    static constexpr auto minimum_copyability = pro::constraint_level::nontrivial;
};

// Client
pro::proxy<MyFacade> p0 = /* Omitted */;
auto p1 = p0; // Calls the constructor of the underlying pointer type
```

In some cases where we clearly know we always instantiate a proxy with a raw pointer, and want to optimize the performance to the limit, it is allowed to add even more constraints in a facade definition, at the cost of reducing the scope of feasible pointer types:

```
// Abstraction
struct MyFacade : pro::facade</* Omitted */> {
    static constexpr auto minimum_copyability = pro::constraint_level::trivial;
    static constexpr auto minimum_relocatability = pro::constraint_level::trivial;
    static constexpr auto minimum_destructibility = pro::constraint_level::trivial;
    static constexpr auto maximum_size = sizeof(void*);
    static constexpr auto maximum_alignment = alignof(void*);
};

// Client
static_assert(std::is_trivially_copy_constructible_v<pro::proxy<MyFacade>>);
static_assert(std::is_trivially_destructible_v<pro::proxy<MyFacade>>);
```

IMPORTANT NOTICE: clang will fail to compile if the `minimum_destructibility` is set to `constraint_level::trivial` in a facade definition. The root cause of this failure is that the implementation requires the language feature defined in P0848R3: Conditionally Trivial Special Member Functions⁷, but it has not been implemented in clang, according to its documentation⁸, at the time this blog was written.

75.11 Highlight 8: Diagnostics

The design of proxy is SFINAE-friendly, thanks to the Concepts feature since C++20. If it is used incorrectly, compile error messages could be generated accurately at the spot. For example, if we call the constructor of proxy with a pointer, whose type does not meet the facade definition:

```
pro::proxy<MyFacade> p;
p.invoke<nullptr_t>(); // nullptr_t is not a valid dispatch type
```

⁷<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0848r3.html>

⁸https://clang.llvm.org/cxx_status.html

Here is the error message gcc 11.2 will report:

```
<source>:550:22: error: no matching function for call to
    'pro::proxy<MyFacade>::invoke<nullptr_t>()'
550 |     p.invoke<nullptr_t>();
|     ~~~~~^~~
<source>:445:18: note: candidate: 'template<class D, class ... Args> decltype(auto)
    pro::proxy<F>::invoke(Args&& ...) requires
(
    pro::details::dependent_traits<
        pro::details::facade_traits<F>, D
    >::dependent_t<
        pro::details::facade_traits<F>, D
    >::applicable
) && (
    pro::details::BasicTraits::has_dispatch<D>
) && (
    is_convertible_v<std::tuple<_Args2 ...>, typename D::argument_types>
) [with D = D; Args = {Args ...}; F = MyFacade] '
445 |     decltype(auto) invoke(Args&&... args)
|           ^~~~~~
<source>:445:18: note:   template argument deduction/substitution failed:
<source>:445:18: note: constraints not satisfied
```

75.12 Conclusion

We hope this has helped clarify how to take advantage of the library “proxy” to write polymorphic code easier. If you have any questions, comments, or issues with the library, you can comment below, file issues in our GitHub repo⁹, or reach us via email at visualcpp@microsoft.com or via Twitter at @VisualC¹⁰.

⁹<https://github.com/microsoft/proxy>

¹⁰ <https://twitter.com/visualc>

Chapter 76

C23 is Finished: Here is What is on the Menu

👤 JeanHeyd Meneide 📅 2022-07-31 💬 ★★★

It's That Blog Post. The release one, where we round up all the last of the features approved since the last time I blogged. If you're new here, you'll want to go check out these previous articles to learn more about what is/isn't going into C23, many of them including examples, explanations, and some rationale:

- C-ing the Improvements¹
- Ever Closer, C23²

The last meeting was pretty jam-packed, and a lot of things made it through at the 11th hour. We also lost quite a few good papers and features too, so they'll have to be reintroduced next cycle, which might take us a whole extra 10 years to do. Some of us are agitating for a faster release cycle, mainly because we have 20+ years of existing practice we've effectively ignored and there's a lot of work we should be doing to reduce that backlog significantly. It's also just no fun waiting for `_attribute_((cleanup(...)))` (defer), statement expressions, better bitfields, wide pointers (a native pointer + size construct), a language-based generic function pointer type (GCC's `void(*)(void)`) and like 20 other things for another 10 years when they've been around for decades.

But, I've digressed long enough: let's not talk about the future, but the present. What's in C23? Well, it's everything (sans the typo-fixes the Project Editors - me 'n' another guy - have to do) present in N3047³. Some of them pretty big blockbuster features for C (C++ will mostly quietly laugh, but that's fine because C is not C++ and we take pride in what we can get done here, with our community.) The first huge thing that will drastically improve code is a combination-punch of papers written by Jens Gustedt and Alex Gilding.

¹<https://thephd.dev/c-the-improvements-june-september-virtual-c-meeting>

²<https://thephd.dev/ever-closer-c23-improvements>

³<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3047.pdf>

76.1 N3006 + N3018 - `constexpr` for Object Definitions

Link⁴.

I suppose most people did not see this one coming down the pipe for C. (Un?)Fortunately, C++ was extremely successful with `constexpr` and C implementations were cranking out larger and larger constant expression parsers for serious speed gains and to do more type checking and semantic analysis at compile-time. Sadly, despite many compilers getting powerful constant expression processors, standard C just continued to give everyone who ended up silently relying on those increasingly beefy and handsome compiler's tricks a gigantic middle finger.

For example, in my last post about C⁵ (or just watching me post on Twitter⁶), I explained how this:

```
const int n = 5 + 4;
int purrs[n];
// ...
```

is some highly illegal contraband. This creates a Variable-Length Array (VLA), not a constant-sized array with size 9. What's worse is that the Usual Compilers™ (GCC, Clang, ICC, MSVC, and most other optimizing compilers actually worth compiling with) were typically powerful enough to basically turn the code generation of this object –so long as you didn't pass it to something expecting an actual Variably-Modified Types (also talked about in another post⁷) –into working like a normal C array.

This left people relying on the fact that this was a C array, even though it never was. And it created enough confusion that we had to accept N2713 to add clarification to the Standard Library to tell people that No, Even If You Can Turn That Into A Constant Expression, You Cannot Treat It Like One For The Sake Of the Language. One way to force an error up-front if the compiler would potentially turn something into not-a-VLA behind-your-back is to do:

```
const int n = 5 + 4;
int purrs[n] = { 0 }; // Bang!
// ...
```

VLAs are not allowed to have initializers¹, so adding one makes a compiler scream at you for daring to write one. Of course, if you're one of those S P E E D junkies, this could waste precious cycles in potentially initializing your data to its 0 bit representation. So, there really was no way to win when dealing with `const` here, despite everyone's mental model –thanks to the word's origin in “constant” –latching onto `n` being a constant expression. Compilers “accidentally” supporting it by either not treating it as a VLA (and requiring the paper I linked to earlier to be added to C23 as a clarification), or treating it as a VLA but extension-ing and efficient-code-generating the problem away just resulted in one too many portability issues. So, in true C fashion, we added a 3rd way that was DEFINITELY unmistakable:

```
constexpr int n = 5 + 4;
int purrs[n] = { 0 }; // Bingo!
// ...
```

⁴<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3018.htm>

⁵<https://thepld.dev/c-the-improvements-june-september-virtual-c-meeting#n2713—integer-constant-expressions-and-their-use-for-arrays>

⁶https://twitter.com/_phantomderp/status/1552424424758022147

⁷<https://thepld.dev/ever-closer-c23-improvements#separating-variably-modified-types-from-variable-length-arrays>

The new `constexpr` keyword for C means you don't have to guess at whether that is a constant expression, or hope your compiler's optimizer or frontend is powerful enough to treat it like one to get the code generation you want if VLAs with other extensions are on-by-default. You are guaranteed that this object is a constant expression, and if it is not the compiler will loudly yell at you. While doing this, the wording for constant expressions was also improved dramatically, allowing:

- compound literals (with the `constexpr` storage class specifier);
- structures and unions with member access by `.;`
- and, the usual arithmetic / integer constant expressions,

to all be constant expressions now.

Oh No, Those Evil Committee People are Ruining™ my Favorite Language®with C++ Nonsense©!

Honestly? I kind of wish I could ruin C sometimes, but believe it or not: we can't!

Note that there are no function calls included in this, so nobody has to flip out or worry that we're going to go the C++ route of stacking on a thousand different "*please make this function constexpr so I can commit compile-time crimes*". It's just for objects right now. There is interest in doing this for functions, but unlike C++ the intent is to provide a level of `constexpr` functions that is so weak it's worse than even the very first C++11 `constexpr` model, and substantially worse than what GCC, Clang, ICC, and MSVC can provide at compile-time right now in their C implementations.

This is to keep it easy to implement evaluation in smaller compilers and prevent feature-creep like the C++ feature. C is also protected from additional feature creep because, unlike C++, there's no template system. What justified half of the improvements to `constexpr` functions in C++ was "*well, if I just rewrite this function in my favorite Functional Language - C++ Templates! - and tax the compiler even harder, I can do exactly what I want with worse compile-time and far more object file bloat*". This was a scary consideration for many on the Committee, but we will not actually go that direction precisely because we are in the C language and not C++.

You cannot look sideways and squint and say "well, if I just write this in the most messed up way possible, I can compute a constant expression in this backdoor Turing complete functional language"; it just doesn't exist in C. Therefore, there is no prior art or justification for an ever-growing selection of constant expression library functions or marked-up headers. Even if we get `constexpr` functions, it will be literally and intentionally be underpowered and weak. It will be so bad that the best you can do with it is write a non-garbage `max` function to use as the behind-the-scenes for a `max` macro with `_Generic`. Or, maybe replace a few macros with something small and tiny.

Some people will look at this and go: "Well. That's crap. The reason I use `constexpr` in my C++-like-C is so I can write beefy compile-time functions to do lots of heavy computation once at a compile-time, and have it up-to-date with the build at the same time. I can really crunch a perfect hash or create a perfect table that is hardware-specific and tailored without needing to drop down to platform-specific tricks. If I can't do that, then what good is this?" And it's a good series of questions, dear reader. But, my response to this for most C programmers yearning for better is this:

we get what we shill for.

With C we do not ultimately have the collective will or implementers brave enough to take-to-task making a large constant expression parser, even if the C language is a lot simpler to write one for compared

to C++. Every day we keep proclaiming C is a simple and beautiful language that doesn't need features, even features that are compile-time only with no runtime overhead. That means, in the future, the only kind of constant functions on the table are ones with no recursion, only one single statement allowed in a function body, plus additional restrictions to get in your way. But that's part of the appeal, right? The compilers may be weak, the code generation may be awful, most of the time you have to abandon actually working in C and instead just use it as a macro assembler and drop down to bespoke, hand-written platform-specific assembly nested in a god-awful compiler-version-specific `#ifdef`, but That's The Close-To-The-Metal C I'm Talkin' About, Babyyyyy!!

"C is simple" also means "the C standard is underpowered and cannot adequately express everything you need to get the job done". But if you ask your vendor nicely and promise them money, cookies, and ice cream, maybe they'll deign to hand you something nice. (But it will be outside the standard, so I hope you're ready to put an expensive ring on your vendor's finger and marry them.)

76.2 N3038 - Introduce Storage Classes for Compound Literals

Link⁸.

Earlier, I sort of glazed over the fact that Compound Literals are now part of things that can be constant expressions. Well, this is the paper that enables such a thing! This is a feature that actually solves a problem C++ was having as well, while also fixing a lot of annoyances with C. For those of you in the dark and who haven't caught up with C99, C has a feature called Compound Literals. It's a way to create any type - usually, structures - that have a longer lifetime than normal and can act as a temporary going into a function. They're used pretty frequently in code examples and stuff done by Andre Weissflog of sokol_gfx.h fame⁹, who writes some pretty beautiful C code (excerpted from the link):

```

1 #define SOKOL_IMPL
2 #define SOKOL_GLCORE33
3 #include <sokol_gfx.h>
4 #define GLFW_INCLUDE_NONE
5 #include <GLFW/glfw3.h>
6
7 int main(int argc, char* argv[]) {
8
9     /* create window and GL context via GLFW */
10    glfwInit();
11    /* ... CODE ELIDED ... */
12
13    /* setup sokol_gfx */
14    sg_setup(&(sg_desc){0}); // Compound Literal
15
16    /* a vertex buffer */
17    const float vertices[] = {
```

⁸<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3038.htm>

⁹https://github.com/flooh/sokol#sokol_gfxh

```

18     // positions           // colors
19     0.0f, 0.5f, 0.5f,    1.0f, 0.0f, 0.0f, 1.0f,
20     0.5f, -0.5f, 0.5f,   0.0f, 1.0f, 0.0f, 1.0f,
21     -0.5f, -0.5f, 0.5f,  0.0f, 0.0f, 1.0f, 1.0f
22 };
23 sg_buffer vbuf = sg_make_buffer(&(sg_buffer_desc){ // ! Compound Literal
24     .data = SG_RANGE(vertices)
25 });
26
27 /* a shader */
28 sg_shader shd = sg_make_shader(&(sg_shader_desc){ // ! Compound Literal
29     .vs.source =
30         "#version 330\n"
31         "layout(location=0) in vec4 position;\n"
32         "layout(location=1) in vec4 color0;\n"
33         "out vec4 color;\n"
34         "void main() {\n"
35             gl_Position = position;\n"
36             color = color0;\n"
37         }\n",
38     .fs.source =
39         "#version 330\n"
40         "in vec4 color;\n"
41         "out vec4 frag_color;\n"
42         "void main() {\n"
43             frag_color = color;\n"
44         }\n"
45 });
46
47 /* ... CODE ELIDED ... */
48 return 0;
49 }

```

C++ doesn't have them (though GCC, Clang, and a few other compilers support them out of necessity). There is a paper by Zhihao Yuan¹⁰ to support Compound Literal syntax in C++, but there was a hang up. Compound Literals have a special lifetime in C called "block scope" lifetime. That is, compound literals in functions behave as-if they are objects created in the enclosing scope, and therefore retain that lifetime. In C++, where we have destructors, unnamed/invisible C++ objects being l-values (objects whose address you can take) and having "Block Scope" lifetime (lifetime until where the next } was) resulted in the usual intuitive behavior of C++'s temporaries-passed-to-functions turning into a nightmare.

For C, this didn't matter and - in many cases - the behavior was even relied on to have longer-lived "temporaries" that survived beyond the duration of a function call to, say, chain with other function calls in

¹⁰<https://wg21.link/p2174>

a macro expression. For C++, this meant that some types of RAII resource holders –like mutexes/locks, or just data holders like dynamic arrays –would hold onto the memory for way too long.

The conclusion from the latest conversation was “we can’t have compound literals, as they are, in C++, since C++ won’t take the semantics of how they work from the C standard in their implementation-defined extensions and none of the implementations want to change behavior” . Which is pretty crappy: taking an extension from C’s syntax and then kind of just…smearing over its semantics is a bit of a rotten thing to do, even if the new semantics are better for C++.

Nevertheless, Jens Gustedt’s paper saves us a lot of the trouble. While default, plain compound literals have “block scope” (C) or “temporary r-value scope” (C++), with the new storage-class specification feature, you can control that. Borrowing the `sg_setup` function above that takes the `sg_desc` structure type:

```
#include <sokol_gfx.h>

SOKOL_GFX_API_DECL void sg_setup(const sg_desc *desc);
```

we are going to add the static modifier, which means that the compound literal we create has static storage duration:

```
int main (int argc, const char* argv[]) {
    /* ... CODE ELIDED ... */
    /* setup sokol_gfx */
    sg_setup(&(static sg_desc){0}); // ! Compound Literal^^I
    /* ... CODE ELIDED ... */
}
```

Similarly, `auto`, `thread_local`, and even `constexpr` can go there. `constexpr` is perhaps the most pertinent to people today, because right now using compound literals in initializers for `const` data is technically SUPER illegal:

```
typedef struct crime {
    int criming;
} crime;

const crime crimes = (crime){ 11 }; // ! ILLEGAL!!

int main (int argc, char* argv[]) {
    return crimes.criming;
}
```

It will work on a lot of compilers (unless warnings/errors are cranked up¹¹), but it’s similar to the VLA situation. The minute a compiler decides to get snooty and picky, they have all the justification in the world because the standard is on their side. With the new `constexpr` specifier, both structures and unions are considered constant expressions, and it can also be applied to compound literals as well:

```
typedef struct crime {
    int criming;
```

¹¹<https://godbolt.org/z/d5Yc6T47a>

```

} crime;

const crime crimes = (constexpr crime){ 11 }; // LEGAL BABYYYYY!

int main (int argc, char* argv[]) {
    return crimes.criming;
}

```

Nice.

76.3 N3017 - #embed

Link¹².

Go read this¹³ to find out all about the feature and how much of a bloody pyrrhic victory it was.

76.4 N3033 - Comma Omission and Deletion (`_VA_OPT_` in C and Preprocessor Wording Improvements)

Link¹⁴.

This paper was a long time coming. C++ got it first, making it slightly hilarious that C harps on standardizing existing practice so much but C++ tends to beat it to the punch for features which solve long-standing Preprocessor shenanigans. If you've ever had to use `_VA_ARGS_` in C, and you needed to pass 0 arguments to that `...`, or try to use a comma before the `_VA_ARGS_`, you know that things got genuinely messed up when that code had to be ported to other platforms. It got a special entry in GCC's documentation¹⁵ because of how blech the situation ended up being:

...GNU CPP permits you to completely omit the variable arguments in this way. In the above examples, the compiler would complain, though since the expansion of the macro still has the extra comma after the format string.

To help solve this problem, CPP behaves specially for variable arguments used with the token paste operator, '`##`'. If instead you write

```
#define debug(format, ...) fprintf (stderr, format, ## _VA_ARGS_)
```

and if the variable arguments are omitted or empty, the '`##`' operator causes the preprocessor to remove the comma before it. If you do provide some variable arguments in your macro invocation, GNU CPP does not complain about the paste operation and instead places the variable arguments after the comma. ...

This is solved by the use of the C++-developed `_VA_OPT_`, which expands out to a legal token sequence if and only if the arguments passed to the variadic `...` are not empty. So, the above could be rewritten as: `#define debug(format, ...) fprintf (stderr, format _VA_OPT_(,) _VA_ARGS_)`

¹²<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3017.htm>

¹³<https://thephd.dev/finally-embed-in-c23>

¹⁴<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3033.htm>

¹⁵<https://gcc.gnu.org/onlinedocs/gcc/Variadic-Macros.html>

This is safe and contains no extensions now. It also avoids any preprocessor undefined behavior. Furthermore, C23 allows you to pass nothing for the `...` argument, giving users a way out of the previous constraint violation and murky implementation behaviors. It works in both the case where you write `debug("meow")` and `debug("meow",)` (with the empty argument passed explicitly). It's a truly elegant design and we have Thomas Köppe to thank for bringing it to both C and C++ for us. This will allow a really nice standard behavior for macros, and is especially good for formatting macros that no longer need to do weird tricks to special-case for having no arguments.

Which, speaking of 0-argument `...` functions...

76.5 N2975 - Relax requirements for variadic parameter lists

Link¹⁶.

This paper is pretty simple. It recognizes that there's really no reason not to allow `void f(...)`; to exist in C. C++ has it, and all the arguments get passed successfully, and nobody's lost any sleep over it. It was also an important filler since, as talked about in old blog posts¹⁷, we have finally taken the older function call style and put it down after 30+ years of being in existence as a feature that never got to see a single proper C standard release non-deprecated. This was great! Except, as that previous blog post mentions, we had no way of having a general-purpose Application Binary Interface (ABI)-defying function call anymore. That turned out to be bad enough that after the deprecation and removal we needed to push for a fix, and lucky for us `void f(...)`; had not made it into standard C yet.

So, we put it in. No longer needing the first parameter, and no longer requiring it for `va_start`, meant we could provide a clean transition path for everyone relying on K&R functions to move to the `...`-based function calls. This means that mechanical upgrades of old codebases - with tools - is now on-the-table for migrating old code to C23-compatibility, while finally putting K&R function calls - and all their lack of safety - in the dirt. 30+ years, but we could finally capitalize on Dennis M. Ritchie's dream here, and put these function calls to bed.

Of course, compilers that support both C and C++, and compilers that already had `void f(...)`; functions as an extension, may have deployed an ABI that is incompatible with the old K&R declarations of `void f()`;. This means that a mechanical upgrade will need to check with their vendors, and:

- make sure that this occupies the same calling convention;
- or, the person who is calling the function cannot update the other side that might be pulling assembly/using a different language,

then the upgrade that goes through to replace every `void f()`; may need to also add a vendor attribute to make sure the function calling convention is compatible with the old K&R one. Personally, I suggest: `[
[vendor::kandr]] void f();`, or something similar. But, ABI exists outside the standard: you'll need to talk to your vendor about that one when you're ready to port to an exclusively-post-C23 world. (I doubt anyone will compile for an exclusively C23-and-above world, but it is nice to know there is a well-defined migration path for users still hook up a 30+ year deprecated feature). Astute readers may notice that if they don't have a parameter to go off of, how do they commit stack-walking sins to get to the arguments? And,

¹⁶<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2975.pdf>

¹⁷<https://thephd.dev/ever-closer-c23-improvements#kr-function-declaration-and-definitions-are->

well, the answer is you still can: ztd.vargs¹⁸ has a proof-of-concept of that (on Windows). You still need some way to get the stack pointer in some cases, but that's been something compilers have provided as an intrinsic for a while now (or something you could do by committing register crimes). In ztd.vargs, I had to drop down into assembly to start fishing for stuff more directly when I couldn't commit more direct built-in compiler crimes. So, this is everyone's chance to get really in touch with that bare-metal they keep bragging about for C. Polish off those dusty manuals and compiler docs, it's time to get intimately familiar with all the sins the platform is doing on the down-low!

76.6 N3029 - Improved Normal Enumerations

Link¹⁹.

What can I say about this paper, except...

What The Hell, Man?

It is absolutely bananas to me that in C –the systems programming language, the language where `const int n = 5` is not a constant expression so people tell you to use `enum { n = 5 }` instead –just had this situation going on, since its inception. “16 bits is enough for everyone” is what Unicode said, and we paid for it by having UTF-16, a maximum limit of 21 bits for our Unicode code points (“Unicode scalar values” if you’re a nerd), and the entire C and C++ standard libraries with respect to text encoding just being completely impossible to use. (On top of the library not working for Big5-HKSCS as a multibyte/narrow encoding). So of course, when I finally sat down with the C standard and read through the thing, noticing that enumeration constants “must be representable by an int” was the exact wording in there was infuriating. 32 bits may be good, but there were plenty of platforms where int was still 16 bits. Worse, if you put code into a compiler where the value was too big, not only would you not get errors on most compilers, you’d sometimes just get straight up miscompiles²⁰. This is not because the compiler vendor is a jerk or bad at their job; the standard literally just phones it in, and every compiler from ICC to MSVC let you go past the low 16-bit limit and occasionally even exceed the 32-bit INT_MAX without so much as a warning. It was a worthless clause in the standard,

and it took a lot out of me to fight to correct this one.

The paper next in this blog post was seen as the fix, and we decided that the old code –the code where people used 0x10000 as a bit flag –was just going to be non-portable garbage. Did you go to a compiler where int is 16 bits and INT_MAX is smaller than 0x10000? Congratulations: your code was non-standard, you’re now in implementation-defined territory, pound sand! It took a lot of convincing, nearly got voted down the first time we took a serious poll on it (just barely scraped by with consensus), but the paper rolled in to C23 at the last meeting. A huge shout out to Aaron Ballman who described this paper as “value-preserving”, which went a really long way in connecting everyone’s understanding of how this was meant to work. It added a very explicit set of rules on how to do the computation of the enumeration constant’s value, so that it was large enough to handle constants like 0x10000 or ULLONG_MAX. It keeps it to be int wherever possible to preserve the semantics of old code, but if someone exceeds the size of int then it’s actually legal to upgrade the backing type now:

```
enum my_values {
```

¹⁸<https://ztdvargs.readthedocs.io/en/latest/>

¹⁹<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3029.htm>

²⁰<https://godbolt.org/z/h5efjs1j1>

```

a = 0, // 'int'
b = 1, // 'int'
c = 3, // 'int'
d = 0x1000, // 'int'
f = 0xFFFFF, // 'int' still
g, // implicit +1, on 16-bit platform upgrades type of the constant here
e = g + 24, // uses "current" type of g - 'long' or 'long long' - to do math and set value
i = ULONG_MAX // 'unsigned long' or 'unsigned long long' now
};


```

When the enumeration is completed (the closing brace), the implementation gets to select a single type that `my_values` is compatible with, and that's the type used for all the enumerations here if int is not big enough to hold ULONG_MAX. That means this next snippet:

```

int main (int argc, char* argv[]) {
    // when enum is complete,
    // it can select any type
    // that it wants, so long as its
    // big enough to represent the type
    return _Generic(a,
        unsigned long: 1,
        unsigned long long: 0,
        default: 3);
}


```

can still return any of 1, 0, or 3. But, at the very least, you know a, or g or i will never truncate or lose the value you put in as a constant expression, which was the goal. The type was always implementation-defined (see: -fshort-enum shenanigans of old). All of that old code that used to be wrong is now no longer wrong. All of those people who tried to write wrappers/shims for OpenGL who used enumerations for their integer-constants-with-nice-identifier-names are also now correct, so long as they are using C23. (This is also one reason why the OpenGL constants in some of the original OpenGL code are written as preprocessor defines (`#define GL_ARB_WHATEVER ...`) and not enumerations. Enumerations would break with any of the OpenGL values above 0xFFFF on embedded platforms; they had to make the move to macros, otherwise it was busted.)

Suffice to say I'm extremely happy this paper got it and that we retroactively fixed a lot of code that **was not supposed to be compiling on a lot of platforms, at all**. The underlying type of an enumeration can still be some implementation-defined integer type, but that's what this next paper is for...

76.7 N3030 - Enhanced Enumerations

Link²¹.

This was the paper everyone was really after. It also got in, and rather than being about “value-preservation”, it was about type preservation. I could write a lot, but whitequark –as usual –describes it best:

²¹<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3030.htm>

i realized today that C is so bad at its job that it needs the help of C++ to make some features of its ABI usable (since you can specify the width of an enum in C++ but not C)

—Catherine (@whitequark), May 25th, 2020²²

C getting dumpstered by C++ is a common occurrence, but honestly? For a feature like this? It is beyond unacceptable that C could not give a specific type for its enumerations, and therefore made the use of enumerations in e.g. bit fields or similar poisonous, bad, and non-portable. There's already so much to contend with in C to write good close-to-the-hardware code: now we can't even use enumerations portably without 5000 static checks special flags to make sure we got the right type for our enumerations? Utter hogwash and a blight on the whole C community that it took this long to fix the problem. But, as whitequark also stated:

in this case the solution to “C is bad at its job” is definitely to “fix C” because, even if you hate C so much you want to eradicate it completely from the face of the earth, we'll still be stuck with the C ABI long after it's gone

—Catherine (@whitequark), May 25th, 2020²³

It was time to roll up my sleeves and do what I always did: take these abominable programming languages to task for their inexcusably poor behavior. The worst part is, I almost let this paper slip by because someone else –Clive Pygott –was handling it. In fact, Clive was handling this even before Catherine made the tweet; N2008, from…

oh my god, it's from 2016²⁴.

I had not realized Clive had been working on it this long until, during one meeting, Clive –when asked about the status of an updated version of this paper –said (paraphrasing) “yeah, I'm not carrying this paper forward anymore, I'm tired, thanks” .

…

That's not, uh, good. I quickly snapped up in my chair, slammed the Mute-Off button, and nearly fumbled the mechanical mute on my microphone as I sputtered a little so I could speak up: “hey, uh, Clive, could you forward me all the feedback for that paper? There's a lot of people that want this feature, and it's really important to them, so send me all the feedback and I'll see if I can do something” . True to Clive's word, minutes after the final day on the mid-2021 meeting, he sent me all the notes. And it was…

…a lot.

I didn't realize Clive had this much push back. It was late 2021. 2022 was around the corner, we were basically out of time to workshop stuff. I frequently went to twitter and ranted about enumerations, from October 2021 and onward. The worst part is, most people didn't know, so they just assumed I was cracked up about something until I pointed them to the words in the standard and then revealed all the non-standard behavior. Truly, the C specification for enumerations was something awful.

Of course, no matter how much I fumed, anger is useless without direction.

I honed that virulent ranting into a weapon: two papers, that eventually became what you're reading about now. N3029 and N3030 was the crystallization of how much I hated this part of C, hated it's specification, loathed the way the Committee worked, and despised a process that led us for over 30 years to end

²²<https://twitter.com/whitequark/status/1265081363717337093>

²³<https://twitter.com/whitequark/status/1265122114811682816>

²⁴<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2008.pdf>

up in this exact spot. This man –Clive –had been at this since 2016. It’s 2022. 5 years in, he gave up trying to placate all the feedback, and that left me only 1 year to clean this stuff up.

Honestly, if I didn’t have a weird righteous anger, the paper would’ve never made it.

Never underestimate the power of anger. A lot of folk and many cultures spend time trying to get you to “manage your emotions” and “find serenity”, often to the complete exclusion of getting mad at things. You wanna know what I think?

““Serenity””

Serenity, peace, all of that can be taken and shoved where the sun don’t shine. We were delivered a hot garbage language, made Clive Pygott –one of the smartest people working on the C Memory Model –gargle Committee feedback for 5 years, get stuck in a rocky specification, and ultimately abandon the effort. Then, we had to do some heroic editing and WAY too much time of 3 people –Robert Seacord, Jens Gustedt, and Joseph Myers –just to hammer it into shape while I had to drag that thing kicking and screaming across the finish line. Even I can’t keep that up for a long time, especially with all the work I also had to do with `#embed` and Modern Bit Utilities and 10+ other proposals I was fighting to fix. “Angry” is quite frankly not a strong enough word to describe a process that can make something so necessary spin its wheels for 5 years. It’s absolutely bananas this is how ISO-based, Committee-based work has to be done. To all the other languages eyeing the mantle of C and C++, thinking that an actual under-ISO working group will provide anything to them.

Do. Not.

Nothing about ISO or IEC or its various subcommittees incentivizes progress. It incentivizes endless feedback loops, heavy weighted processes, individual burn out, and low return-on-investment. Do anything –literally anything –else with your time. If you need the ISO sticker because you want ASIL A/B/C/D certification for your language, than by all means: figure out a way to make it work. But keep your core process, your core feedback, your core identity out of ISO. You can standardize existing practices way better than this, and without nearly this much gnashing of teeth and pullback. No matter how politely its structured, the policies of ISO and the way it expects Committees to be structured is a deeply-embedded form of bureaucratic violence against the least of these, its contributors, and you deserve better than this. So much of this CIA sabotage field manual’s list²⁵:

should not have a directly-applicable analogue that describes how an International Standards Organization conducts business. But if it is what it is, then it’s time to roll up the sleeves. Don’t be sad. Get mad. Get even²⁶.

Anyways, enumerations. You can add types to them:

```
enum e : unsigned short {
    x
};

int main (int argc, char* argv[]) {
    return _Generic(x, unsigned short: 0, default: 1);
}
```

Unlike before, this will always return 0 on every platform, no exceptions. You can stick it in structures

²⁵<https://www.openculture.com/2022/01/read-the-cias-simple-sabotage-field-manual.html>

²⁶<https://www.youtube.com/watch?v=AM0zMexL6ok>

- (1) Insist on doing everything through "channels." Never permit short-cuts to be taken in order to, expedite decisions.
- (2) Make "speeches." Talk as frequently as possible and at great length. Illustrate your "points" by long anecdotes and accounts of personal experiences. Never hesitate to make a few appropriate "patriotic" comments.
- (3) When possible, refer all matters to committees, for "further study and consideration." Attempt to make the committees as large as possible.
- (4) Bring up irrelevant issues as frequently as possible.
- (5) Haggle over precise wordings of communications, minutes, resolutions.
- (6) Refer back to matters decided upon at the last meeting and attempt to reopen the question of the advisability of that decision.
- (7) Advocate "caution." Be "reasonable" and urge your fellow-conferees to be "reasonable" and avoid haste which might result in embarrassments or difficulties later on.
- (8) Be worried about the propriety of any decision -raise the question of whether such action as is contemplated lies within the jurisdiction of the group or whether it might conflict with the policy of some higher echelon.

and unions and use it with bitfields and as long as your implementation is not completely off its rocker, you will get entirely dependable alignment, padding, and sizing behavior. Enjoy!

76.8 N3020 - Qualifier-preserving Standard Functions

Link²⁷.

This is a relatively simple paper, but closes up a hole that's existed for a while. Nominally, it's undefined-behavior to modify an originally-const array – especially a string literal – through a non-const pointer. So,

why exactly was `strchr`, `bsearch`, `strpbrk`, `strrchr`, `strstr`, `memchr`, and their wide counterparts basically taking const in and stripping it out in the return value?

The reason is because these had to be singular functions that defined a single externally-visible function call. There's no overloading in C, so back in the old days when these functions were cooked up, you could only have one. We could not exclude people who wanted to write into the returned pointers of these functions, so we made the optimal (at the time) choice of simply removing the const from the return values. This was not ideal, but it got us through the door.

Now, with type-generic macros in the table, we do not have this limitation. It was just a matter of someone getting inventive enough and writing the specification up for it, and that's exactly what Alex Gilding did! It looks a little funny in the standardese, but:

```
#include <string.h>
QChar *strchr(QChar *s, int c);
```

²⁷<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3020.pdf>

Describes that if you pass in a const-qualified char, you get back a const-qualified char. Similarly if there is no const. It's a nice little addition that can help improve read-only memory safety. It might mean that people using any one of the aforementioned functions as a free-and-clear "UB-cast" to trick the compiler will have to fess up and use a real cast instead.

76.9 N3042 - Introduce the nullptr constant

Link <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3042.htm>.

To me, this one was a bit obviously in need, though not everyone thinks so. For a long time, people liked using NULL, (void*)0, and literal 0 as the null pointer constant. And they are certainly not wrong to do so: the first one in that list is a preprocessor macro resolving to either of the other 2. While nominally it would be nice if it resolved to the first, compatibility for older C library implementations and the code built on top of it demands that we not change NULL. Of course, this made for some interesting problems in portability:

```
#include <stdio.h>

int main (int argc, char* argv[]) {
    printf("ptr: %p", NULL); // oops
    return 0;
}
```

Now, nobody's passing NULL directly to printf(...), but in a roundabout way we had NULL - the macro itself - filtering down into function calls with variadic arguments. Or, more critically, we had people just passing straight up literal 0. "It's the null pointer constant, that's perfectly fine to pass to something expecting a pointer, right?" This was, of course, wrong. It would be nice if this was true, but it wasn't, and on certain ABIs that had consequences. The same registers and stack locations for passing a pointer were not always the same as were used for literal 0 or - worse - they were the same, but the literal 0 didn't fill in all the expected space of the register (32-bit vs. 64-bit, for example). That meant people doing printf("%p", 0); in many ways were relying purely on the luck of their implementation that it wasn't producing actual undefined behavior! Whoops.

nullptr and the associated nullptr_t type in <stddef.h> fixes that problem. You can specify nullptr, and it's required to have the same underlying representation as the null pointer constant in char* or void* form. This means it will always be passed correctly, for all ABIs, and you won't read garbage bits. It also aids in the case of _Generic: with NULL being implementation-defined, you could end up with void* or 0. With nullptr, you get exactly nullptr_t: this means you don't need to lose the _Generic slot for both int or void*, especially if you're expecting actual void* pointers that point to stuff. Small addition, gets rid of some Undefined Behavior cases, nice change.

Someone recently challenged me, however: they said this change is not necessary and bollocks, and we should simply force everyone to define NULL to be void*. I said that if they'd like that, then they should go to those vendors themselves and ask them to change and see how it goes. They said they would, and they'd like a list of vendors defining NULL to be 0. Problem: quite a few of them are proprietary, so here's my Open Challenge:

if you (yes, you!!) have got a C standard library (or shim/replacement) where you define NULL to be 0 and not the void-pointer version, send me a mail and I'll get this person in touch with you so you can duke it out with each other. If they manage to convince enough vendors/maintainers, I'll convince the National Body I'm with to write a National Body Comment asking for nullptr to be rescinded. Of course, they'll need to not only reach out to these people, but convince them to change their NULL from 0 to ((void*)0), which. Well.

Good luck to the person who signed up for this.

76.10 N3022 - Modern Bit Utilities

Link²⁸.

Remember how there were all those instructions available since like 1978 –you know, in the Before Times™, before I was even born and my parents were still young? –and how we had easy access to them through all our C compilers because we quickly standardized existing practice from last century?

…Yeah, I don't remember us doing that either.

Modern Bit Utilities isn't so much "modern" as "catching up to 40-50 years ago". There were some specification problems and I spent way too much time fighting on so many fronts that, eventually, something had to suffer: although the paper provides wording for Rotate Left/Right, 8-bit Endian-Aware Loads/Stores, and 8-bit Memory Reversal (fancy way of saying, "byteswap"), the specification had too many tiny issues in it that opposition mounted to prevent it from being included-and-then-fixed-up-during-the-C23-commenting-period, or just included at all. I was also too tired by the last meeting day, Friday, to actually try to fight hard for it, so even though a few other members of WG14 sacrificed 30 minutes of their block to get Rotate Left/Right in, others insisted that they wanted to do the Rotate Left/Right functions in a different style. I was too tired to fight too hard over it, so I decided to just defer it to post-C23 and come back later.

Sorry.

Still, with the new <stdbit.h>, this paper provides:

- Endian macros (`__STDC_ENDIAN_BIG__`, `__STDC_ENDIAN_LITTLE__`, `__STDC_ENDIAN_NATIVE__`)
- `stdc_popcount`
- `stdc_bit_width`
- `stdc_leading_zeroes/stdc_leading_ones/stdc_trailing_zeros/stdc_trailing_ones`
- `stdc_first_leading_zero/stdc_first_leading_one/stdc_first_trailing_zero/stdc_first_trailing_one`
- `stdc_has_single_bit`
- `stdc_bit_width`
- `stdc_bit_ceil`
- `stdc_bit_floor`

²⁸<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3022.htm>

“Where’s the endian macros for Honeywell architectures or PDP endianness?” You can get that if `__STDC_ENDIAN_NATIVE__` isn’t equal to either the little OR the big macro:

```
#include <stdbit.h>
#include <stdio.h>

int main (int argc, char* argv[]) {
    if (__STDC_ENDIAN_NATIVE__ == __STDC_ENDIAN_LITTLE__) {
        printf("little endian! uwu\n");
    }
    else if (__STDC_ENDIAN_NATIVE__ == __STDC_ENDIAN_BIG__) {
        printf("big endian OwO!\n");
    }
    else {
        printf("what is this?!\n");
    }
    return 0;
}
```

If you fall into the last branch, you have some weird endianness. We do not provide a macro for that name because there is too much confusion around what the exact proper byte order for “PDP Endian” or “Honeywell Endian” or “Bi Endian” would end up being.

“What’s that ugly `stdc_` prefix?”

For the bit functions, a prefix was added to them in the form of `stdc_`... Why?

`popcount` is a really popular function name. If the standard were to take it, we’d effectively be loading up a gun to shoot a ton of existing codebases right in the face. The only proper resolution I could get to the problem was adding `stdc_` in front. It’s not ideal, but honestly it’s the best I could do on short notice. We do not have namespaces in C, which means any time we add functionality we basically have to square off with users. It’s most certainly not a fun part of proposal development, for sure: thus, we get a `stdc_` prefix. Perhaps it will be the first of many functions to use such prefixes so we do not have to step on user’s toes, but I imagine for enhancements and fixes to existing functionality, we will keep writing function names by the old rules. This will be decided later by a policy paper, but that policy paper only applies to papers after C23 (and after we get to have that discussion).

76.11 N3006 + N3007 - Type Inference for object definitions

Link²⁹.

This is a pretty simple paper, all things considered. If you ever used `__auto_type` from GCC: this is that, with the name `auto`. I describe it like this because it’s explicitly not like C++’s `auto` feature: it’s significantly weaker and far more limited. Whereas C++’s `auto` allows you to declare multiple variables on the same line and even deduce partial qualifiers / types with it (such as `auto* ptr1 = thing, *ptr2 = other_thing;` to demand that `thing` and `other_thing` are some kind of pointer or convertible to one), the

²⁹<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3007.htm>

C version of `auto` is modeled pretty directly after the weaker version of `auto type`. You can only declare one variable at a time. There's no pointer-capturing. And so on, and so forth:

```
int main (int argc, char* argv[]) {
    auto a = 1;
    return a; // returns int, no mismatches
}
```

It's most useful in macro expressions, where you can avoid having to duplicate expressions with:

```
#define F(_NAME, ARG, ARG2, ARG3) \
    typeof(ARG + (ARG2 || ARG3)) _NAME = ARG + (ARG2 | ARG3);

int main (int argc, char* argv[]) {
    F(a, 1, 2, 3);
    return a;
}
```

instead being written as:

```
#define F(_NAME, ARG, ARG2, ARG3) \
    auto _NAME = ARG + (ARG2 | ARG3);

int main (int argc, char* argv[]) {
    F(a, 1, 2, 3);
    return a;
}
```

Being less prone to make subtle or small errors that may not be caught by the compiler you're using is good, when it comes to working with specific expressions. (You'll notice the left hand side of the `_NAME` definition in the first version had a subtle typo. If you did: congratulations! If you didn't: well, `auto` is for you.) Expressions in macros can get exceedingly complicated, and worse if there are unnamed structs or similar being used it can be hard-to-impossible to name them. `auto` makes it possible to grasp these types and use them properly, resulting in a smoother experience.

Despite being a simple feature, I expect this will be one of the most divisive for C programmers. People already took to the streets in a few places to declare C a dead language, permanently ruined by this change. And, as a Committee member, if that actually ends up being the case? If this actually ends up completely destroying C for any of the reasons people have against `auto` and type inference for a language that quite literally just let you completely elide types in function calls and gave you “implicit `int`” behavior that compilers today still have to support so that things like OpenSSL can still compile?²

Don't threaten me with a good time, now.

76.12 N2897 - memset_explicit

Link³⁰.

³⁰<https://open-std.org/jtc1/sc22/wg14/www/docs/n2897.htm>

`memset_explicit` is `memset_s` from Annex K, without the Annex K history/baggage. It serves functionally the same purpose, too. It took a lot (perhaps too much) discussion, but Miguel Ojeda pursued it all the way to the end. So, now we have a standard, mandated, always-present `memset_explicit` that can be used in security-sensitive contexts, provided your compiler and standard library implementers work together to not Be Evil™.

Hoorah!

76.13 N2888 - Exact-width Integer Types May Exceed (u)intmax_t

Link³¹.

The writing has been on the wall for well over a decade now; `intmax_t` and `uintmax_t` have been inadequate for the entire industry over and has been consistently limiting the evolution of C's integer types year over year³², and affecting downstream languages. While we cannot exempt every single integer type from the trappings of `intmax_t` and `uintmax_t`, we can at least bless the `intN_t` types and `uintN_t` types so they can go beyond what the two max types handle. There is active work in this area to allow us to transition to a better ABI and let these two types live up to their promises, but for now the least we could do is let the vector extensions and extended compiler modes for `uint128_t`, `uint256_t`, `uint512_t`, etc. etc. all get some time in the sun and out of the (u)`intmax_t` shadow.

This doesn't help for the preprocessor, though, since you are still stuck with the maximum value that `intmax_t` and `uintmax_t` can handle. Integer literals and expressions will still be stuck dealing with this problem, but at the very least there should be some small amount of portability between the Beefy Machines™ and the presence of the newer `UINT128_WIDTH` and such macros.

Not the best we can do, but progress in the right direction!

76.14 And That's All I'm Writing About For Now

Note that I did not say "that is it" : there's quite a few more features that made it in, just my hands are tired and there's a lot of papers that were accepted. I also do not feel like there are some I can do great justice with, and quite frankly the papers themselves make better explanations than I do. Particularly, N2956 - unsequenced functions³³ is a really interesting paper that can enable some intense optimizations with user attribute markup. Its performance improves can also be applied locally:

```

1 #include <math.h>
2 #include <fenv.h>
3
4 inline double distance (double const x[static 2]) [[reproducible]] {
5     #pragma FP_CONTRACT OFF
6     #pragma FENV_ROUND FE_TONEAREST
7     // We assert that sqrt will not be called with invalid arguments
8     // and the result only depends on the argument value.
9     extern typeof(sqrt) [[unsequenced]] sqrt;
```

³¹<https://open-std.org/jtc1/sc22/wg14/www/docs/n2888.htm>

³²<https://thephd.dev/binary-banshees-digital-demons-abi-c-c++-help-me-god-please#abi-even-simpler>

³³<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2956.htm>

```

10     return sqrt(x[0]*x[0] + x[1]*x[1]);
11 }
```

I'll leave the paper to explain how exactly that's supposed to work, though! On top of that, we also removed Trigraphs??!(N2940)³⁴ from C, and we made it so the `_BitInt` feature³⁵ can be used with bit fields (N2969, nice)³⁶. (If you don't know what Trigraphs are, consider yourself blessed.)

Another really consequential paper is the Tag Compatibility paper by Martin Uecker, N3037³⁷. It makes for defining generic data structures through macros a lot easier, and does not require a pre-declaration in order to use it nicely. A lot of people were thrilled about this one and picked up on the improvement immediately³⁸: it helps us get one step closer to maybe having room to start shipping some cool container libraries in the future. You should be on the lookout for when compilers implement this, and rush off to the races to start developing nicer generic container libraries for C in conjunction with all the new features we put in!

There is also a lot of functionality that did not make it, such as Unicode Functions, defer, Lambdas/Blocks/Nested Functions³⁹, wide function pointers, `constexpr` functions, the `byteswap` and other low-level bit functionality I spoke of before, statement expressions, additional macro functionality, `break` `break` (or something like it), `size_t` literals, `__supports_literal`, Transparent Aliases⁴⁰, and more.

But For Now? My work is done. I've got to go take a break and relax. You can find the latest draft copy of the Committee Draft Standard N3047⁴¹ here. It's probably filled with typos and other mistakes; I'm not a great project editor, honestly, but I do try, and I guess that's all I can do for all of us. That's it for me and C for the whole year. So now, it's sleepy time. Nighty night, and thanks for coming on this wild ride with me.

³⁴<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2940.pdf>

³⁵<https://thephd.dev/c-the-improvements-june-september-virtual-c-meeting#n2709---adding-a-fundamental-type-for-n-bit-integers>

³⁶<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2969.htm>

³⁷<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3037.pdf>

³⁸https://www.reddit.com/r/C_Programming/comments/w5hl80/comment/ih8jxi6/?context=3

³⁹<https://thephd.dev/lambdas-nested-functions-block-expressions-oh-my>

⁴⁰<https://thephd.dev/to-save-c-we-must-save-abi-fixing-c-function-abi>

⁴¹<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3047.pdf>

Chapter 77

Cpp2 and cppfront: Year-end mini-update

👤 Herb Sutter 📅 2022-12-31 💬 ★★★

As we close out 2022, I thought I’d write a short update on what’s been happening in Cpp2 and cppfront¹. If you don’t know what this personal project is, please see the CppCon 2022 talk on YouTube².

Most of this post is about improvements I’ve been making and merging over the year-end holidays, and an increasing number of contributions from others via pull requests to the cppfront repo and in companion projects. Thanks again to so many of you who expressed your interest and support for this personal experiment, including the over 3,000 comments on Reddit and YouTube and the over 200 issues and PRs on the cppfront repo!

77.1 10 design notes

On the cppfront wiki³, I’ve written more design notes about specific parts of the Cpp2 language design that answer common questions. They include:

- Broad strategic topics, such as addressing ABI and versioning, “unsafe” code, and aiming to eliminate the preprocessor with reflection.
- Specific language feature design topics, such as unified function call syntax (UFCS), const, and namespaces.
- Syntactic choices, such as postfix operators and capture syntax.
- Implementation topics, such as parsing strategies and grammar details.

¹<https://github.com/hsutter/cppfront>

²<https://www.youtube.com/watch?v=ELeZAKCN4tY>

³<https://github.com/hsutter/cppfront/wiki>

77.2 117 issues (3 open), 74 pull requests (9 open), 6 related projects, and new collaborators

I started cppfront with “just a blank text editor and the C++ standard library.” Cppfront continues to have no dependencies on other libraries, but since I open-sourced the project in September I’ve found that people have started contributing working code —thank you! Authors of merged pull requests include:

- The prolific **Filip Sajdak** contributed a number of improvements, probably the most important being generalizing my UFCS implementation, implementing more of `is` and `as` as described in P2392⁴, and providing Apple-Clang regression test results. Thanks, Filip!
- **Gabriel Gerlero** contributed refinements in the Cpp2 language support library, `cpp2util.h`.
- **Jarosław Glowacki** contributed test improvements and ensuring all the code compiles cleanly at high warning levels on all major compilers.
- **Konstantin Akimov** contributed command-line usability improvements and more test improvements.
- **Fernando Pelliccioni** contributed improvements to the Cpp2 language support library.
- **Jessy De Lannoit** contributed improvements to the documentation.

Thanks also to these six related projects, which you can find listed on the wiki⁵:

- Conan recipe for cppfront⁶ by Fernando Pelliccioni.
- `/modern-cmake/cppfront`⁷, a modern CMake (3.23+) build for cppfront.
- `/JohelEGP/jegp.cmake_modules/#jegpcpp2`⁸, a CMake module to build Cpp2 source files.
- Meson support for cppfront⁹ by Jussi Pakkanen, creator of Meson.
- Sublime syntax highlighting for Cpp2¹⁰ by Andrew Feldman.
- Visual Studio Code syntax highlighting for Cpp2¹¹ by Elazar Cohen. (I use this by default.)

Thanks again to **Matt Godbolt** for hosting cppfront on Godbolt Compiler Explorer and giving feedback.

Thanks also to over 100 other people who reported bugs and made suggestions via the Issues. See below for some more details about these features and more.

⁴<https://wg21.link/p2392>

⁵<https://github.com/hsutter/cppfront/wiki>

⁶<https://github.com/conan-io/conan-center-index/tree/master/recipes/cppfront>

⁷<https://github.com/modern-cmake/cppfront>

⁸https://github.com/JohelEGP/jegp.cmake_modules/#jegpcpp2

⁹<https://nibblestew.blogspot.com/2022/10/using-cppfront-with-meson.html>

¹⁰<https://github.com/12Thanjo/cppfront-sublime>

¹¹<https://marketplace.visualstudio.com/items?itemName=elazarcoh.cpp2-syntax>

77.3 Compiler/language improvements

Here are some highlights of things added to the cppfront compiler since I gave the first Cpp2 and cppfront talk in September¹². Most of these were implemented by me, but some were implemented by the PR authors I mentioned above.

Roughly in commit order (you can find the whole commit history here¹³), and like everything else in cppfront some of these continue to be experimental:

- Lots of bug fixes and diagnostic improvements.
- Everything compiles cleanly under MSVC `-W4` and GCC/Clang `-Wall -Wextra`.
- **Enabled implicit move-from-last-use for all local variables.** As I already did for `copy` parameters.
- **After repeated user requests, I turned `-n` and `-s` (null/subscript dynamic checking) on by default.** Yes, you can always still opt out to disable them and get zero cost, Cpp2 will always stay a “zero-overhead don’t-pay-for-what-you-don’t-use” true-C++ environment. All I did was change the default to enable them.
- **Support explicit forward of members/subobjects of composite types.** For a parameter declared `forward x: SomeType`, the default continues to be that the last use of `x` is automatically forwarded for you; for example, if the last use is `call_something(x)`; then cppfront automatically emits that call as `call_something(std::forward<decltype(x)>(x))`; and you never have to write out that incantation. But now you also have the option to separately forward parts of a composite variable, such as that for a `forward x: pair<string, string>>` parameter you can write things like `do_this(forward x.first)` and `do_that(1, 2, 3, forward x.second)`.
- Support `is template-name` and `is ValueOrPredicate`: `is` now supports asking whether this is an instantiation of a template (e.g., `x is std::vector`), and it supports comparing values (e.g., `x is 14`) and using predicates (e.g., `x is (less_than(20))` invoking a lambda) including for values inside a `std::variant`, `std::any`, and `std::optional` (e.g., `x is 42` where `x` is a `variant<int,string>` or an `any`).
- **Regression test results for all major compilers:** MSVC, GCC, Clang, and Apple-Clang. All are now checked in and can be conveniently compared before each commit.
- **Finished support for » and »= expressions.** In today’s syntax, C++ currently max-munches the `>` and `»` tokens and then situationally breaks off individual `>` tokens, so that we can write things like `vector<vector<int>>` without putting a space between the two closing angle brackets. In Cpp2 I took the opposite choice, which was to not parse `>or »` as a token (so max munch is not an issue), and just merge closing angles where a `>or »` can grammatically go. I’ve now finished the latter, and this should be done.
- **Generalized support for UFCS.** In September, I had only implemented UFCS for a single call of the form `x.f(y)`, where `x` could not be a qualified name or have template arguments. Thanks to Filip Sajdak for generalizing this to qualified names, templated names, and chaining multiple UFCS calls! That was a lot of work, and as far as I can tell UFCS should now be generally complete.

¹²<https://www.youtube.com/watch?v=ELeZAKCN4tY>

¹³<https://github.com/hsutter/cppfront/commits/main>

- **Support declaring multi-level pointers/const.**
- **Zero-cost implementation of UFCS.** The implementation of UFCS is now force-inlined on all compilers. In the tests I've looked at, even when calling a nonmember function `f(x,y)`, using Cpp2's `x.f(y)` unified function call syntax (which tries a member function first if there is one, else falls back to a nonmember function), the generated object code at all optimization levels is now identical, or occasionally better, compared to calling the nonmember function directly. Thanks to Pierre Renaux for pointing this out¹⁴!
- **Support today's C++ (Cpp1) multi-token fundamental types (e.g., signed long long int).** I added these mainly for compatibility because 100
- **Support fixed-width integer type aliases (i32, u64, etc.),** including optional `_fast` and `_small` (e.g., `i32_fast`).

I think that this completes the basic implementation of Cpp2's initial subset that I showed in my talk in September, including that support for multi-level pointers and the multi-word C/C++ fundamental type names should complete support for being able to invoke any existing C and C++ code seamlessly.

Which brings us to…

77.4 What's next

Next, as I said in the talk, I'll be adding support for user-defined types (classes)…I'll post an update about that when there's more that's ready to see.

Again, thanks to everyone who expressed interest and support for this personal experiment, and may you all have a happy and safe 2023.

¹⁴<https://github.com/hsutter/cppfront/issues/185>

Chapter 78

Copy-Paste Developments

👤 Jonathan Bocvara 📅 2022-04-26 💬 ★★

Amongst the many tasks a programmer does, one of them is to add a new feature in a location of the application where there are already many similar existing features.

The temptation is then to warm up very specific muscles of our left hand:

- the pinky muscles that will press on the **Ctrl** key,
- the index finger muscles to press on the **C** key
- whatever muscles on the right-hand side of our index finger that will move it over the **V** key.

In other words, we prepare ourselves for a **copy-paste** development. That is, we find something in the application that is similar to what we want to add, copy-paste it, and change the small bits that are specific to our new feature.

Although the nervous and muscular biological sequence behind a copy-paste is beautiful, there is another beautiful nervous element we can use: our brain.

Indeed, even in copy-paste development, injecting a bit of understanding can reap a lot of benefits, both for our satisfaction and for the quality of our code.

78.1 Copy-paste developments

First let's agree on what we call copy-paste developments. I'm assuming that you follow the DRY (Don't Repeat Yourself) principle, that is trying to avoid code duplication, at least when this is reasonable¹.

So copy-pasting is not your default approach. But in some developments, where the feature you're adding is similar to many existing features that are all copy-pasted from one another, it can be difficult not to copy-paste your new one into existence.

For example, you're asked to add a new field in a dictionary of existing fields. Or a new value in a report. Or a new field in an XML that is produced by your application.

And you're not very familiar with the framework of that dictionary, or with that reporting module, or that XML library.

¹<https://www.fluentcpp.com/2018/11/13/to-dry-or-not-to-dry/>

Fortunately, there are many fields in the dictionary, many values in the report, or many fields in the XML, and they all kind of look alike, except for some specific bits to each field.

Have you ever been asked to do such a development?

The temptation can be to just copy-paste, ignore what you don't really understand, modify the specific part for your new field, and call it a day.

But there is a better way.

78.2 The need to understand

We generally don't see those developments as very interesting or rewarding. But there is one thing we can do to make them more interesting and turn them into learning opportunities: **understanding the code we're copy-pasting**.

Of course, not having to copy-paste any code because the framework you're working in is very well designed is the ideal situation. But if you're working with existing code, sometimes you don't choose the quality of the code, or at least the quality it has when you start working with it.

But the least we can do is to understand the code we're copy-pasting.

Here are several benefits this can bring.

78.3 Dead code

One reason to understand what you're copy-pasting is that perhaps part of it is not necessary for your specific feature.

This has happened to me in a recent development, and realising it saved me from introducing a substantial amount of code inspired from another feature. The most amusing part is that I realised that this other feature I took as a model didn't need that part too!

Introducing useless code is sad. It makes the code less expressive because there is more to read and understand for the next person. And we don't get anything for it, because it's useless.

Understanding what you're copy-pasting, even if it's a vague understanding about what each part is used for, can make your code simpler and more expressive.

78.4 You'll need to understand it anyway

In general, when you add a new feature in legacy code, does it work the first time after you've compiled it and run the application?

Sometimes it does. But sometimes, we need to adjust the code to make it work and cover all cases. If we don't understand how the code works, we're crippled to solve its issues and make our new feature work.

The worst situation is when we think it works, and a bug is discovered later. Then we need to get back to the unfamiliar copy-pasted code and figure it out. And if the bug is urgent to fix because it has been uncovered late in the process...this is not a situation we want to be in, is it?

Since we're likely going to figure out the code at some point anyway, let's do it at the beginning and write the code as correctly as possible as early in the process as possible.

78.5 Extending your known scope

Taking the time to understand a new framework or module expands your knowledge, and in software engineering, and in legacy code in particular, Knowledge is Power.

For a lot more details about this, you can check out chapter 7 of The Legacy Code Programmer's Toolbox².

78.6 Intellectual interest

A lot of us became developers because they enjoy the intellectual challenge in it. But copy-pasting is a pretty dumb activity.

Personally, the part of programming that I enjoy most is understanding how existing systems work. Even understanding how copy-pasted code work brings me more satisfaction than copy-pasting mysterious code and hoping for it to work.

Is this the same for you? What part of your job do you enjoy the most? Let me know in the comments section!

I wouldn't presume of what brings you intellectual satisfaction in your job as a programmer, but I'd love to know if understanding systems bring you joy too.

78.7 Not copy-pasting

If you understand the code you're copy-pasting, you'll be more able to see a common pattern between the various features.

This can give you the tools to put some code in common and copy-paste less code to implement your feature, thus applying the DRY principle.

At some point you may even be knowledgeable enough about the patterns used by the framework you're inserting your code in to refactor the framework itself, in a separate development.

This is great exercise, both in detecting abstractions in code and in formulating new ones. This is fundamental, as programming all comes down to respecting levels of abstraction³.

78.8 Don't let your fingers muscles do all the work

Do copy-paste developments sound familiar?

When you have to do one, do you invest the time and effort to understand the copy-pasted code?

If you don't, next time you have to do a copy-paste development, why not use your full brain power to understand the code you're copy-pasting, and see what the benefits this can bring you?

Share your experience in the comments below!

²<https://leanpub.com/legacycode/>

³<https://www.fluentcpp.com/2016/12/15/respect-levels-of-abstraction/>

Chapter 79

Carbon's most exciting feature is its calling convention

👤 Jonathan 📅 2022-07-29 💬 ★★★

Last week, Chandler Carruth announced Carbon¹, a potential C++ replacement they've been working on for the past two years. It has the usual cool features you expect from a modern language: useful generics, compile-time interfaces/traits/concepts, modules, etc. –but the thing I'm most excited about is a tiny detail about the way parameters are passed there.

It's something I've been thinking about in the past myself, and to my knowledge it hasn't been done in any low-level language before, but the concept has a lot of potential. Let me explain what I'm talking about.

79.1 Carbon's parameter passing

By default, i.e. if you don't write anything else, Carbon parameters are passed by the equivalent of a `const T&` in C++.

```
1 class Point
2 {
3     var x: i64;
4     var y: i64;
5     var z: i64;
6 }
7
8 fn Print(p : Point);

1 struct Point
2 {
3     std::uint64_t x, y, z;
4 };
```

¹<https://github.com/carbon-language/carbon-lang>

```

5
6 void Print(const Point& p);

```

However – and this is the import part – the compiler is allowed to convert that to a T under the as-if rule².

```

fn Print(x : i32);

void Print(std::int32_t x);

```

... and so what? Why am I so excited about that?

79.2 Advantage #1: Performance

Passing things by `const` T& is always good, right? After all, you’re avoiding a copy!

While true, references are essentially pointers on the assembly level. This means that passing an argument by `const` T& sets a register to its address, which means

1. in the caller, the argument needs an address and must be stored in memory somewhere, and
2. in the callee, the parameter needs to load the value from memory when its read.

This is the only options for types that don’t fit in a register, or small types with non-trivial copy constructors, but it’s less ideal for trivially copyable types that do fit.

Compare the assembly(<https://godbolt.org/z/qTrP3oMT3>) between the add function that takes its arguments by `const` T&

```

1 [[gnu::noinline]] int add(const int& a, const int& b)
2 {
3     return a + b;
4 }
5
6 int foo()
7 {
8     return add(11, 42);
9 }

```

and the one that doesn’t

```

1 [[gnu::noinline]] int add(int a, int b)
2 {
3     return a + b;
4 }
5
6 int foo()
7 {
8     return add(11, 42);
9 }

```

²https://en.cppreference.com/w/cpp/language/as_if

All the memory stores and loads just disappear; you don't want to be passing `int`'s by reference!

So it's really nice that in Carbon you don't need to think about it - the compiler will just do the correct thing for you. Furthermore, you can't always do it manually.

79.3 Advantage #2: Optimal calling convention in generic code

Suppose we want to write a generic function print function in C++. The type can be arbitrarily large with an arbitrarily expensive copy constructor, so the you should use `const T&` in generic code.

```
template <typename T>
void Print(const T& obj);
```

However, this pessimizes the situation for small and cheap types, which is unfortunate. It's also not something the compiler can fix with optimizations, because the function signature and calling convention is part of the -here comes our favorite three-letter acronym -ABI. At best, the compiler can inline it and elide the entire call.

There are ways to work around that problem³, because of course there are, but it just works™ in Carbon, which is nice.

But the real reason I'm excited about the feature has nothing to do with eliding memory load/stores.

79.4 Advantage #3: Copies that aren't copies

Note that the transformation the compiler can do isn't quite the same as `const T& -> T` in C++ would do. The latter creates a copy of the argument: if needed, it will invoke the copy constructor and destructor.

In Carbon, this isn't the case: the value is simply set to a register. As the called function does not call the destructor of the parameter, the caller does not need to call the copy constructor. This means that the optimization would even be valid for Carbon's equivalent of `std::unique_ptr`. The caller simply sets a register to the underlying pointer value, and the callee can access it. No transfer of ownership happens here.

This isn't something you can do in (standard) C++.

79.5 Advantage #4: Parameters without address

If you've been thinking about the consequences of that language feature, you might wonder about Carbon code like the following:

```
1 fn Print(p : Point)
2 {
3     var ptr : Point* = &p;
4     ...
5 }
```

If the compiler decides to pass `p` in a register, you can't create a pointer to it. So the code doesn't compile - you must not take the address of a parameter (unless its declared using the `var` keyword).

³https://www.boost.org/doc/libs/1_79_0/libs/utility/doc/html/utility/utilities/call_traits.html

Without additional annotation, parameters of a Carbon function do not expose their address to the compiler, as they might not have any. This is the real reason I’m so excited about that feature.

79.6 More precise escape analysis

Since a programmer can’t take the address of a parameter, escape analysis⁴ does not need to consider them. For example, in the following C++ code, what is returned by the function?

```

1 void take_by_ref(const int& i);
2
3 void do_sth();
4
5 int foo()
6 {
7     int i = 0;
8     take_by_ref(i);
9     i = 11;
10    do_sth();
11    return i;
12 }
```

Well, 11 right?

However, the following is a valid implementation of `take_by_ref()` and `do_sth()`:

```

1 int* ptr; // global variable
2
3 void take_by_ref(const int& i)
4 {
5     // i wasn't const to begin with, so it's fine
6     ptr = &const_cast<int&>(i);
7 }
8
9 void do_sth()
10 {
11     *ptr = 42;
12 }
```

Suddenly, `foo()` returns 42 –and this was 100% valid. As such, the compiler has to separately reload the value stored in `i` before returning, it escapes.

In Carbon, this is impossible, `take_by_ref()` can’t sneakily store the address somewhere where it can come back to haunt you. As such, `i` will not escape and the compiler can optimize the function to return 11.

⁴https://en.wikipedia.org/wiki/Escape_analysis

79.7 Explicit address syntax

Is the following C++ code okay?

```

1  class Widget
2  {
3      public:
4          void DoSth(const std::string& str);
5      };
6
7  Widget Foo()
8  {
9      Widget result;
10
11     std::string str = "Hello!";
12     result.DoSth(str);
13
14     return result;
15 }
```

It depends.

`Widget::DoSth()` can get the address of the function-local string and store it somewhere. Then when its returned from the function, it contains a dangling pointer.

In Carbon, this is impossible –if `widget` wants to store a pointer, it needs to accept a pointer:

```

1  class Widget
2  {
3      fn DoSth[addr me : Self*](str : String*);
4  }
```

Crucially, calling code then also needs to take the address:

```

1  fn Foo() -> Widget
2  {
3      var result : Widget;
4
5      var str : String = "Hello";
6      result.DoSth(&str);
7
8      return result;
9  }
```

The extra syntax in the call makes it really obvious that something problematic might be going on here. For the same reason, the Google C++ style guide used to require pointers⁵ in C++ code in such situations.

⁵<https://stackoverflow.com/questions/26441220/googles-style-guide-about-input-output-parameters-as-pointers>

This has the unfortunate side-effect that you can pass `nullptr` to the parameter, so I've suggested in the past to use my `type_safe::object_ref` instead⁶.

Tips

This situation also makes it clear that references aren't simply non-null pointers, which is a common misconception. References and pointers have crucial differences.^a

^a<https://www.jonathanmueller.dev/talk/cppnow2018/>

79.8 Future language extensions

Disclaimer: I'm not a Carbon developer, I'm just someone with opinions.

In parameters, `foo : T` is a parameter whose address can't be taken, and `var foo : T` is a parameter with an address. The same principle can also be applied to more situations. For example, consider the following classes:

```

1  class Birthday
2  {
3      var year : i32;
4      var month : i8;
5      var day : i8;
6  }
7
8  class Person
9  {
10     var birthday : Birthday;
11     var number_of_children : i8;
12 }
```

I know, it's silly. Bear with me.

Assuming Carbon follows the same rules for data layout, as C++ the size of `Birthday` is 8 bytes (4 bytes for `year`, 1 for `month`, 1 for `day` and 2 padding bytes at the end), and the size of `Person` is 12 bytes (8 bytes for `Birthday`, 1 byte for `number_of_children`, and 3 for padding).

A more optimal layout would eliminate `Birthday` and inline the members into `Person`:

```

1  class Person
2  {
3      var birthday_year : i32;
4      var birthday_month : i8;
5      var birthday_day : i8;
6      var number_of_children : i8;
7 }
```

^bhttps://github.com/foonathan/type_safe

Now, the size of Person is only 8 bytes because `number_of_children` can be stored in what were padding bytes before.

Is this an optimization the compiler could do?

Not really, because it needs to preserve a separate `Birthday` subobject: someone could take the address of the `birthday` member and pass it around.

Tips

While it could work here, because we're just stuffing things into padding bytes; in general optimal layout might require splitting an existing subobject into two different parts or shuffling the members around differently. Then there simply exists no contiguous sequence of bytes that make up the member, so there no pointer to it can exist.

However, we could imagine member variables where you can't take the address, signified by a lack of `var`:

```

1 class Person
2 {
3     birthday : Birthday;
4     number_of_children : i8;
5 }
```

Now the compiler is free to change the layout, inline struct members and shuffle them around. Note that taking the address of `birthday.month` (and the other members) is still fine: it's been declared with `var` and its stored contiguously in memory – just not necessarily next to `year` and `day`. `var` and non-`var` members can be freely mixed.

Similarly, an optimization that transforms Array of Structs to Struct of Arrays⁷ is also invalid, as in the first layout you have each individual struct in one contiguous chunk of memory that have an address, but in the second the struct members have been split. If you have an array where you can't take the address of elements however, this isn't something you can observe.

Finally, extending it to local variables essentially enables the `register` keyword⁸ from C: local variables without an address that can safely live in registers. While it isn't necessary for modern optimizers, it's still less work if the compiler doesn't need to consider them during escape analysis at all. More importantly, it documents intent to the reader.

79.9 Conclusion

Creating entities whose address can't be taken is a simple feature with lots of potential. It enables many optimizations to change layout, as layout can't be observed, it simplifies escape analysis and optimizes parameter passing.

It's also not really a limitation in many cases: how often do you actually need to take the address of something? Marking those few situations with an extra keyword doesn't cost you anything.

⁷https://en.wikipedia.org/wiki/AoS_and_SoA

⁸[https://en.wikipedia.org/wiki/Register_\(keyword\)](https://en.wikipedia.org/wiki/Register_(keyword))

I really wish C++ had it as well, but it wouldn't work with functions that take references, which makes them useless unless the language was designed around it from the start.

This is exactly where Carbon comes in.

Chapter 80

Ways to Refactor Toggle/Boolean Parameters in C++

▀ Bartłomiej Filipek 📅 2022-02-21 ▣★★★

Boolean parameters in a function might be misleading and decrease its readability. If you have a poorly named function like:

```
DoImportantStuff(true, false, true, false);
```

As you can imagine, it's not clear what all those parameters mean? What's the first `true`? What does the last `false` mean? Can we improve code in such cases?

Let's have a look at possible improvements.

80.1 Intro

This article was motivated by a similar text that appeared on Andrzej Krzemienski's Blog: Toggles in functions¹.

As Andrzej² wrote, the whole point is to improve the code around functions like:

```
RenderGlyphs(glyphs, true, false, true, false);
```

What if you mix two parameters and change their order? The compiler won't help you much!

Let's think about improving the code: make it safer and more readable.

We could add comments:

```
RenderGlyphs(glyphs,
    /*useCache*/true,
    /*deferred*/false,
    /*optimize*/true,
    /*finalRender*/false);
```

And although the above code is a bit more readable, we still don't get any more safety.

Can we do more?

¹<https://akrzemil.wordpress.com/2017/02/16/toggles-in-functions/>

²<https://akrzemil.wordpress.com/about/>

80.2 ideas

Here are some ideas that you can use to make such code better.

80.2.1 Small enums

We could write the following declarations:

```
enum class UseCacheFlag { False, True };
enum class DeferredFlag { False, True };
enum class OptimizeFlag { False, True };
enum class FinalRenderFlag { False, True };

// and call like:
RenderGlyphs(glyphs,
              UseCacheFlag::True,
              DeferredFlag::False,
              OptimizeFlag::True,
              FinalRenderFlag::False);
```

And in the implementation you need to change:

```
if (useCache) { }
else { }
if (deferred) { }
else {}
```

To proper comparison:

```
if (useCache == UseCacheFlag::True) { }
else { }
if (deferred == DeferredFlag::True) { }
else {}
```

As you can see, you need to check against enum values rather than just check the bool value.

Using enums is a good approach, but it has some disadvantages:

- A lot of additional names are required! Maybe we could reuse some types. Should we have some common flags defined in the project? How to organize those types?
- Values are not directly convertible to bool, so you have to compare against Flag::True explicitly inside the function body.

The required explicit comparison was the reason Andrzej wrote his own little library³ that creates toggles with conversion to bool. I was disappointed that we don't have direct support from the language for strong types for enums. But after a while, I changed my mind. The explicit comparison is not that hard to write, so maybe it would be overkill to include it in the language spec? Introducing explicit casts might even cause some problems.

Still, I am not entirely happy with the need to write so many tiny enums...

³https://github.com/akrzemil/explicit/blob/master/include/ak_toolkit/tagged_bool.hpp

80.2.2 Bit flags

As a potential evolution for enums, you can also use bit flags.

Unfortunately, we don't have friendly and type-safe support from the language, so you need to add some boilerplate code to support all operations.

Here's my simplified approach:

```

1 #include <type_traits>
2
3 struct Glyphs { };
4
5 enum class RenderGlyphsFlags
6 {
7     useCache = 1,
8     deferred = 2,
9     optimize = 4,
10    finalRender = 8,
11 };
12
13 // simplification...
14 RenderGlyphsFlags operator | (RenderGlyphsFlags a, RenderGlyphsFlags b) {
15     using T = std::underlying_type_t <RenderGlyphsFlags>;
16     return static_cast<RenderGlyphsFlags>(static_cast<T>(a) | static_cast<T>(b));
17     // todo: missing check if the new value is in range...
18 }
19
20 constexpr bool IsSet(RenderGlyphsFlags val, RenderGlyphsFlags check) {
21     using T = std::underlying_type_t <RenderGlyphsFlags>;
22     return static_cast<T>(val) & static_cast<T>(check);
23     // todo: missing additional checks...
24 }
25
26 void RenderGlyphs(Glyphs &glyphs, RenderGlyphsFlags flags)
27 {
28     if (IsSet(flags, RenderGlyphsFlags::useCache)) { }
29     else { }
30
31     if (IsSet(flags, RenderGlyphsFlags::deferred)) { }
32     else { }
33
34     // ...
35 }
36
37 int main() {

```

```

38     Glyphs glyphs;
39     RenderGlyphs(glyphs, RenderGlyphsFlags::useCache | RenderGlyphsFlags::optimize);
40 }
```

What do you think about this approach? With some additional code and operator overloading, we can end up with a nice function that is readable and typesafe. If you add more checks into my example code, you can enforce that the values you pass have the right bit set.

80.2.3 Param structure

If you have several parameters (like 4 or 5, depending on the context), why don't we wrap them into a separate structure?

```

1  struct RenderGlyphsParam
2  {
3      bool useCache;
4      bool deferred;
5      bool optimize;
6      bool finalRender;
7  };
8  void RenderGlyphs(Glyphs &glyphs, const RenderGlyphsParam &renderParam);
9
10 // the call:
11 RenderGlyphs(glyphs,
12             /*useCache*/true,
13             /*deferred*/false,
14             /*optimize*/true,
15             /*finalRender*/false});
```

OK...this didn't help much! You get additional code to manage, and the caller uses almost the same code.

Yet, this approach has the following advantages:

- It moves the problem to the other place. You could apply strong types to individual members of the structure.
- If you need to add more parameters, you can just extend the structure.
- Especially useful if more functions can share such param structure.

Side note: you could put the glyphs variable also in the RenderGlyphsParam, this is only for example.

80.2.4 How about C++20

Thanks to Designated Initializers that landed in C++20, we can use “named” parameters when constructing our small structure.

Basically, you could use a similar approach as in C99 and name arguments that you pass to a function:

```

1 struct RenderGlyphsParam
2 {
3     bool useCache;
4     bool deferred;
5     bool optimize;
6     bool finalRender;
7 };
8 void RenderGlyphs(Glyphs &glyphs, const RenderGlyphsParam &renderParam);
9
10 // the call:
11 RenderGlyphs(glyphs,
12             {.useCache = true,
13              .deferred = false,
14              .optimize = true,
15              .finalRender = false});
```

You can read my blog post on this new feature here: Designated Initializers in C++20 - C++ Stories⁴.

80.2.5 Elimination

We could try to fix the syntax and use clever techniques. But what about using a simpler method? What if we provide more functions and just eliminate the parameter?

It's OK to have one or two toggle parameters, but if you have more, maybe it means a function tries to do too much?

In our simple example, we could try the split in the following way:

```

RenderGlyphsDeferred(glyphs,
                     /*useCache*/true,
                     /*optimize*/true);
RenderGlyphsForFinalRender(glyphs,
                          /*useCache*/true,
                          /*optimize*/true);
```

We can make the change for parameters that are mutually exclusive. In our example, deferred cannot happen together with the final run.

You might have some internal function RenderGlyphsInternal that would still take those toggle parameters (if you really cannot separate the code). But at least such internal code will be hidden from the public API. You can refactor that internal function later if possible.

I think it's good to look at the function declaration and review if there are mutually exclusive parameters. Maybe the function is doing too much? If yes, then cut it into several smaller functions.

After writing this section, I've noticed a tip from Martin Fowler on Flag Arguments⁵. In the text, he also tries to avoid toggles.

⁴<https://www.cppstories.com/2021/designated-init-cpp20/>

⁵<https://martinfowler.com/bliki/FlagArgument.html>

You can also read this article from Robert C. Martin's Clean Code Tip #12: Eliminate Boolean Arguments⁶. And more in his book Clean Code: A Handbook of Agile Software Craftsmanship⁷

80.2.6 Stronger types

Using small enums or structures is a part of a more general topic of using Stronger Types. Similar problems might appear when you have several ints as parameters or strings...

You can read more about:

- Strong Types in C++: A Concrete Example - C++ Stories⁸
- Simplify C++: Use Stronger Types! -⁹
- Type safe handles in C++ —I Like Big Bits¹⁰
- Strong types for strong interfaces - Fluent C++¹¹
- foonathan::blog() - Type safe - Zero overhead utilities for more type safety¹²
- Serialization - BOOST_STATIC_WARNING¹³

80.2.7 C++ Guidelines

Thankfully we also have C++ Guidelines, and we can reach for help here.

There's an item: I.4: Make interfaces precisely and strongly typed¹⁴ which not only talks about boolean parameters but all sorts of potentially misleading names.

For instance, the guidelines mention the following cases:

```
draw_rect(100, 200, 100, 500); // what do the numbers specify?
```

```
draw_rect(p.x, p.y, 10, 20); // what units are 10 and 20 in?
```

As an improvement, we can use the following approaches:

- Pass a separate structure so that the arguments converts into data members
- Consider using a flags enum
- Consider using strong types, for example passing `std::chrono::milliseconds` rather than `int num_msec` to a function.

What's more, as potential enforcement from the code analysis tools they suggest:

Look for functions that use too many primitive type arguments

⁶<http://www.informit.com/articles/article.aspx?p=1392524>

⁷<http://amzn.to/2m3g2LS>

⁸<https://www.cppstories.com/2021/strong-types-pesel/>

⁹<https://arne-mertz.de/2016/11/stronger-types/>

¹⁰<http://www.ilikebigbits.com/blog/2014/5/6/type-safe-identifiers-in-c>

¹¹<http://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/>

¹²<http://foonathan.net/blog/2016/10/11/type-safe.html>

¹³http://www.boost.org/doc/libs/1_61_0/libs/serialization/doc/strong_TYPEDEF.html

¹⁴<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#i4-make-interfaces-precisely-and-strongly-typed>

80.2.8 Tools

Speaking of tooling, one reader¹⁵ suggested that in Clang-Tidy there's a check that enforces “named parameters comments” near the arguments.

This feature is called: clang-tidy - bugprone-argument-comment —Extra Clang Tools 15.0.0git documentation¹⁶.

For example:

```
void RenderGlyphs(Glyphs &glyphs,
    bool useCache, bool deferred, bool optimize, bool finalRender, int bpp)
{

}

int main() {
    Glyphs glyphs;
    RenderGlyphs(glyphs,
        /*useCha=*/
        /*deferred=*/
        /*optimize=*/
        /*finalRender=*/
        /*bpppp=*/
        /*8);
}
}
```

You'll get the following message:

```
<source>:13:14: warning: argument name 'useCha' in comment does not
match parameter name 'useCache' [bugprone-argument-comment]
    /*useCha=*/
    ^
<source>:5:8: note: 'useCache' declared here
    bool useCache, bool deferred, bool optimize, bool finalRender, int bpp)
```

80.3 A concrete example

Recently, I had a chance to apply some ideas of enum/stronger types to my code. Here's a rough outline:

```
// functions:
bool CreateContainer(Container *pOutContainer, bool *pOutWasReused);

void Process(Container *pContainer, bool bWasReused);
```

¹⁵https://www.reddit.com/r/cpp/comments/sxpsxt/comment/hxtweel/?utm_source=share&utm_medium=web2x&context=3

¹⁶<https://clang.llvm.org/extralang-tools/extra/clang-tidy/checks/bugprone-argument-comment.html>

```
// usage
bool bWasReused = false;
if (!CreateContainer(&myContainer, &bWasReused))
    return false;

Process(&myContainer, bWasReused);
```

Briefly: we create a container, and we process it. The container might be reused (through a pool, reusing existing objects, etc., some internal logic).

I thought that it didn't look nice. We use one output flag, and then it's passed as input to some other function.

What's more, we pass pointers, and some additional validation should happen. Also, the output parameters are discouraged in Modern C++, so it's not good to have them anyway.

How can we do better?

Let's use enums!

```
enum class ContainerCreateInfo { Err, Created, Reused };
ContainerCreateInfo CreateContainer(Container *pOutContainer);

void Process(Container *pContainer, ContainerCreateInfo createInfo);

// usage
auto createInfo = CreateContainer(&myContainer)
if (createInfo == ContainerCreateInfo::Err);
    return false;

Process(&myContainer, createInfo);
```

Isn't it better?

There are no outputs via pointer stuff here; we have a strong type for the 'toggle' parameter.

Also, if you need to pass some more information in that CreateInfo enum, you can just add one more enum value and process it in proper places; the function prototypes don't have to change.

Of course, in the implementation, you have to compare against enum values (not just cast to bool), but it is not difficult and even more verbose.

Is that all?

The code is still not perfect as I have pOutContainer, which is not ideal.

In my real project, that was a complex thing to change, and I wanted to reuse existing containers... But if your container support move semantics and you can rely on Return Value Optimization, then it's possible to return it:

```
enum class ContainerCreateInfo { Err, Created, Reused };
std::pair<Container, ContainerCreateInfo> CreateContainer();
```

Our function becomes a factory function, but it has to return some additional information about the creation process.

We can use it as follows:

```
// usage
auto [myContainer, createInfo] = CreateContainer()
if (createInfo == ContainerCreateInfo::Err);
    return false;

Process(&myContainer, createInfo);
```

80.4 Summary

By reading the original article from Andrzej¹⁷ and these additional few words from me, I hope you get the idea about toggle type parameters. They are not totally wrong, and it's probably impossible to avoid them entirely. Still, it's better to review your design when you want to add third or fourth parameter in a row :) Maybe you can reduce the number of toggles/flags and have more expressive code?

¹⁷<https://www.cppstories.com/2017/03/on-toggle-parameters/>&<https://akrzemiel.wordpress.com/2017/02/16/toggles-in-functions/>

Chapter 81

malloc() and free() are a bad API

👤 Jonathan 📅 2022-08-31 🎵 ★★☆

If you need to allocate dynamic memory in C, you use `malloc()` and `free()`. The API is very old, and while you might want to switch to a different implementation, be it *jemalloc*, *tcmalloc*, or *mimalloc*, they mostly copy the interface. It makes sense that they do that –they want to be a mostly drop-in replacement, but it’s still unfortunate because `malloc()` and `free()` are a bad API for memory allocation.

Let’s talk why.

81.1 The C allocation functions

`malloc()` and `free()` have a very simple interface: `malloc()` takes a size and returns a pointer to the allocated memory block of that size, `free()` takes a previously allocated pointer and frees it.

```
void* malloc(size_t size);
void free(void* ptr);
```

Then there is also `calloc()`, which allocates memory that has been zeroed. For whatever reason, it has a slightly different interface:

```
void* calloc(size_t num, size_t size);
```

Logically, it allocates `num` objects of `size` each, i.e. `num * size` bytes. It also does an overflow check for you, because why not.

Finally, there is `realloc()`:

```
void* realloc(void* ptr, size_t new_size);
```

It attempts to grow or shrink a memory block to the `new_size`. This may or may not copy things around in memory, and returns the new starting address, or `ptr` unchanged if it was left in-place. Notably, `malloc()` can be implemented in terms of `realloc()`:

```
void* malloc(size_t size)
{
    return realloc(NULL, size);
}
```

Seems straightforward enough, what’s the issue?

81.2 Problem #1: Alignment

Plain old `malloc()` does not allow specifying a custom alignment for the aligned memory. It returns memory that is suitable aligned for any object with fundamental alignment.

Want to allocate a SIMD vector or something aligned at a page boundary? It gets tricky:

```

1  constexpr auto page_size = 4096;
2
3  void* allocate_page_boundary(std::size_t size)
4  {
5      // Allocate extra space to guarantee alignment.
6      auto memory = std::malloc(page_size + size);
7
8      // Align the starting address.
9      auto address = reinterpret_cast<std::uintptr_t>(memory);
10     auto misaligned = address & (page_size - 1);
11
12     return static_cast<unsigned char*>(memory) + page_size - misaligned;
13 }
```

Of course, you can't free the resulting address with `std::free()`, since it may point somewhere inside the allocated memory block. You have to remember the original address as well.

For example, you can store it in the alignment padding directly preceding the aligned address.

At least C11 has added `aligned_alloc()`, which then became part of C++17:

```
void* aligned_alloc(size_t alignment, size_t size);
```

This doesn't help you with `realloc()` or `calloc()`, however.

81.3 Problem #2: Metadata storage

`malloc()` doesn't directly go ahead and ask the OS for memory, that would be too slow. Instead there are various caches for memory blocks of varying sizes.

For example, a program often allocates 8 byte elements, so it might make sense to keep a list of 8 byte blocks. When you ask for 8 bytes, it simply returns one from the list:

```

1  void* malloc(size_t size)
2  {
3      if (size == 8)
4          return block_list_8_bytes.pop();
5
6      ...
7 }
```

Then when we free an 8 byte memory block, it is added to the list instead:

```

1 void free(void* ptr)
2 {
3     if (size_of_memory(ptr) == 8)
4     {
5         block_list_8_bytes.push(ptr);
6         return;
7     }
8
9     ...
10 }
```

Of course this requires that the allocator knows the size of a memory block given its pointer. The only way to do that is to store some metadata about the allocator somewhere. This could be a global hash table that maps pointers to sizes, or extra metadata store directly in front of the address, as discussed in the over aligned example. In either case, it means that asking for 8 bytes of memory will not actually allocate 8 bytes of memory, but additional metadata as well.

This is especially wasteful because the user usually knows how big the memory block is it's currently attempting to free!

```

1 template <typename T>
2 class dynamic_array
3 {
4     T* ptr;
5     std::size_t size;
6
7 public:
8     explicit dynamic_array(T* ptr, std::size_t size)
9         : ptr(static_cast<T*>(std::malloc(size * sizeof(T))))
10    {}
11
12     ~dynamic_array()
13    {
14         ... // call destructors
15
16         // I know that I'm freeing size * sizeof(T) bytes!
17         std::free(ptr);
18    }
19};
```

If `free()` took the size of the memory block as extra parameter, the implementation wouldn't need to add extra metadata just for that.

Of course an implementation might choose to allocate extra metadata for performance reasons anyway. Currently it's forced to do that.

81.4 Problem #3: Wasting space

Consider the implementation of `std::vector<T>::push_back()`. When there is no capacity to store an additional element it needs to reserve bigger memory and move everything over. To keep an amortized O(1) complexity, it grows the new memory by some factor:

```

1 void push_back(const T& obj)
2 {
3     if (size() == capacity())
4     {
5         auto new_capacity = std::max(2 * capacity(), 1);
6         auto new_memory = std::malloc(new_capacity * sizeof(T));
7
8         ...
9     }
10
11     ...
12 }
```

This works, but can waste memory.

Suppose the implementation of `std::malloc` uses a cache of recently freed memory blocks. When attempting to allocate **N** blocks, it searches that cache for a block that is at least **N** bytes big. If it finds one (either the first one that fits, or the smallest one that fits, or ...), returns it. In that case, the returned memory block might have space for more than **N** bytes!

This means that we ask for a memory with a capacity for e.g. 14 elements, but get a memory block with a capacity for 16 elements instead. But we don't know that! We treat the block as if it has space for 14 elements only and trigger another unnecessary reallocation for the 15th element.

It would be great if `std::malloc()` could return how big the allocated memory block actually is, so we can leverage any extra space we might have gotten “for free” .

81.5 Problem #4: realloc()

`realloc()` attempts to grow a memory block in-place. If that's not possible, it allocates a new one and copies the existing contents over. This is done as-if by `std::memcpy()`.

This automatic copy is problematic.

For starters, it can't be used with C++ objects that might want to invoke a move constructor. It also doesn't work with C objects that have self-referential pointers such as a buffer containing a circular linked list.

This is a shame as `realloc()`'s ability to grow a memory block in-place is really useful and not achievable in any other way. Sadly, it can't be used with e.g. `std::vector`.

81.6 A better interface

Let me propose a new interface that doesn't have those shortcomings. It consists of three functions `allocate()`, `deallocate()`, and `try_expand()`.

`allocate()` is the replacement for `std::malloc()`. Its goal is to allocate a memory block for a given size and alignment. Crucially it returns not only a pointer to the allocated memory, but also the total size that is available for the user.

```

1 struct memory_block
2 {
3     void* ptr;
4     size_t size;
5 };
6
7 /// On success `result.ptr != NULL` and `result.size >= size`.
8 /// On failure, `result.ptr == NULL` and `result.size == 0`.
9 memory_block allocate(size_t size, size_t alignment);

```

This takes care of problem #1 and #3.

`deallocate()` is a replacement for `std::free()`. It takes a `memory_block` as well, in addition to the alignment that was used to request this block:

```
void deallocate(memory_block block, size_t alignment);
```

That way, we pass all information the caller has anyway to the allocator.

It might make sense to relax the requirements for `deallocate()` a bit, so that the size of the block doesn't need to be the exact value returned by `allocate()`, but some other value between the requested size and the actual size. For example, a `std::vector<int>` might request 12 bytes (3 ints), but gets 17 bytes (4 ints plus one spare). It would then call `deallocate()` with a size of 16 bytes, as it knows the capacity in number of ints.

Finally, `try_expand()` is a replacement for `realloc()`. Crucially, it will only attempt to expand the block in-place, and fail if that's not possible.

```

/// If the block can be expanded in-place to `new_size`, returns true.
/// Otherwise, returns `false`.
bool try_expand(memory_block block, size_t new_size);

```

This solves problem #4 by making the caller responsible for copying the allocated memory if necessary.

81.7 C++ Solutions

C++'s `operator new` and `operator delete`, have inherited the same problems:

```

void* operator new(std::size_t size);
void operator delete(void* ptr);

```

```
// not pictured: dozens of other overloads
```

To its credit, it keeps making improvements.

81.8 C++17: Aligned allocation

C++17 adds an overload that accepts `std::align_val_t`, which allows specification of a custom alignment.

```
void* operator new(std::size_t size, std::align_val_t alignment);
void operator delete(void* ptr, std::align_val_t alignment);
```

The alignment is passed as `std::align_val_t` because a user might have overloaded `operator new` (`std::size_t`, `std::size_t`) already.

81.9 C++17: Sized deallocation

A user can actually define its own implementation of `operator new/delete` to control all memory allocations. This is then invoked by the compiler to allocate memory. Since C++17, the compiler will also attempt to invoke the following overloads:

```
void operator delete(void* ptr, std::size_t size);
void operator delete(void* ptr, std::size_t size, std::align_val_t alignment);
```

As the compiler knows the size of the objects its deallocating, it can pass that information to the function. If you’re writing a custom allocator implementation, you don’t need to worry about metadata.

Of course, this doesn’t help the default implementation using `std::malloc` and `std::free`.

81.10 C++23: Size feedback in `std::allocator`

C++23 has adopted P0401, which adds a new function to `std::allocator`:

```
1 template<class Pointer>
2 struct allocation_result
3 {
4     Pointer ptr;
5     size_t count;
6 };
7
8 class allocator
9 {
10 public:
11     allocation_result<T*> allocate_at_least(size_t n);
12 };
```

The function does what it says: it allocates memory for at least n objects and returns the actual size of the memory available. This behaves like my proposed `allocate()` function.

The language side with changes for `operator new` as proposed by P0901 is still in the standardization process, and will hopefully come in C++26.

81.11 Conclusion

A good API requests all information it needs (duh) and returns as much information it can provide (law of useful return). `malloc()` and `free()` don't follow those principles, which make them less useful as they could be.

It's great to see that C++23 has finally fixed most of those shortcomings, at least on the library side. Of course, modern languages like Rust don't make any of the mistakes in the first place¹.

¹<https://doc.rust-lang.org/alloc/alloc/traitAllocator.html>

Chapter 82

Use compiler flags for stack protection in GCC and Clang

• Serge Guelton, Siddhesh Poyarekar  2022-06-02 

A binary may be subject to a wide range of attacks, but smashing the stack for fun and profit¹ is one of the most venerable ones. Stack-based attacks are also the lowest-cost form of attack: Stack layouts are quite predictable because data containing return addresses can be overwritten to gain control over program execution.

Compiler developers have implemented a wide range of countermeasures in response. This article discusses the major stack protection mechanisms in the GNU Compiler Collection (GCC) and Clang, typical attack scenarios, and the compiler's attempts to prevent attacks.

82.1 Stack canary

A *stack canary* is the most rudimentary check for buffer overflows on the stack. The canary is an extra word of memory at the end of the stack frame with a value set at runtime. This value is presumably unknown to the attacker and checked for modification before jumping out of the function. A modification indicates the detection of a stack smashing followed by a termination routine.

Stack canaries are added by GCC and Clang through these flags:

- `-fstack-protector`
- `-fstack-protector-strong`
- `-fstack-protector-all`
- `-fstack-protector-explicit`

82.2 SafeStack and shadow stack

This form of protection splits the stack into two distinct areas, storing precious variables and user variables in non-contiguous memory areas. The goal is to make it more difficult to smash one of the stacks

¹https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

from the other. This process occurs in hardware (e.g., x86_64 shadow stack) or in software such as LLVM's SafeStack.

GCC and Clang add shadow stack through this flag:

- `-mshstk`

Clang adds SafeStack through this flag:

- `-fsanitize=safe-stack`

82.3 Fortified source

The GNU C library (glibc) provides alternate implementations of some commonly used functions to smash the stack by copying a given amount of bytes from one address to another. These implementations typically use compiler support to check for memory bounds of objects. If an operation overflows these bounds, it would cause programs to terminate. This defense is called FORTIFY_SOURCE and the Red Hat blog has an excellent post on the subject².

GCC and Clang select fortified sources using the preprocessor flag `-D_FORTIFY_SOURCE=<n>` with an optimization level greater than `-O0`. A higher `<n>` corresponds to greater protection. However, there may be a tradeoff in performance, and some programs that conform to the standard may fail due to stricter security checks. GCC (version 12 and later) and Clang (version 9.0 and later) support `<n>` up to 3.

82.4 Control flow integrity

Return-oriented programming³ (ROP) uses an initial stack smash to take control of an indirect jump and then executes an arbitrary sequence of instructions. One countermeasure to this kind of attack is to ensure that jump addresses and return addresses are correct by using hardware support or pure software.

GCC and Clang can generate support code for Intel's Control-flow Enforcement Technology⁴ (CET) through this compiler flag:

- `-fcf-protection=[full]`

GCC and Clang can also generate support code for Branch Target Identification⁵ (BTI) on AArch64 using:

- `-mbranch-protection=none|bt1|pac-ret+leaf|pac-ret[+leaf+b-key]|standard`

In addition to these flags, which require hardware support, Clang provides a software implementation of control flow integrity.

- `-fsanitize=cfi`

²<https://www.redhat.com/en/blog/security-technologies-fortifysource>

³<https://www.redhat.com/en/blog/fighting-exploits-control-flow-integrity-cfi-clang>

⁴<https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html>

⁵<https://developer.arm.com/documentation/ddi0596/2020-12/Base-Instructions/BTI-Branch-Target-Identification>

82.5 Stack allocation control

The following options have an impact on the stack allocation. These options are not necessarily designed to provide extra security, but they may be a nice side-effect.

On an x86 target, GCC and Clang provide the ability to automatically allocate a discontiguous stack when running out of stack memory. The `-fsplit-stack` activates this behavior.

When using the GNU linker, it is possible to pass a stack limit to the program, reading from a symbol or a register. This process is supported by GCC using:

- `-fstack-limit-register`
- `-fstack-limit-symbol`

These stack-allocated arrays are a common entry point for a stack-based attack. The Clang compiler makes it possible to reduce the attack surface by disallowing this pattern, subsequently preventing the use of stack-allocated arrays. The code `-fno-stack-array` enables this process.

82.6 Stack usage and statistics

Most stack usage consists of either call stack information or fixed-sized allocations in the form of local variables. There is, however, a facility to allocate dynamically sized objects on the stack by using the `alloca()` function or using variable-length arrays (VLAs). `alloca()` usage with sizes can be user-controlled. This is a common target for overflowing stacks or making them clash with other maps in memory. As a result, a well-behaved application must always keep track of them. Both GCC and Clang provide ways to control `alloca()` and stack usage to prevent clashes at the outset.

The following flags allow finer control over stack usage in applications and provide a warning when `alloca()` or VLA cross developer-defined thresholds:

- `-Wframe-larger-than`
- `-Walloca`
- `-Walloca-larger-than`
- `-Wvla -Wvla-larger-than`

There are also recursion checks as well as higher-level checks on stack usage to help developers get better control of the stack behavior.

Here is a full list of warnings implemented in GCC⁶:

- `-Wstack-usage`
- `-fstack-usage`
- `-Walloca`
- `-Walloca-larger-than`

⁶<https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc/Invoking-GCC.html#Invoking-GCC>

- `-Wvla`
- `-Wvla-larger-than`
- `-Wframe-larger-than`
- `-mwarn-dynamicstack (s390x)`
- `-Wstack-protector`
- `-Wtrampolines`
- `-Winfinite-recursion`

And here is a list of the warnings implemented in Clang⁷:

- `-fstack-usage`
- `-Walloca`
- `-Wframe-larger-than`
- `-Wstack-usage`

82.7 Summary

Both GCC and Clang provide a wide range of compiler flags to prevent stack-based attacks. Some of these flags relate to a specific kind of exploit. Others introduce generic protection. And some flags give feedback like warnings and reports to the user, providing a better understanding of the behavior of the stack program. Depending on the attack scenario, code size constraints, and execution speed, compilers provide a wide range of tools to address the attack.

⁷<https://blog.quarkslab.com/clang-hardening-cheat-sheet.html>

Chapter 83

The Wonderfully Terrible World of C and C++ Encoding APIs (with Some Rust)

👤 JeanHeyd Meneide 📅 2022-10-12 💬 ★★★★

Last time we talked about encodings, we went in with a C++-like design where we proved that so long as you implement the required operations on a single encoding type, you can go between any two encodings on the planet. This meant you didn't need to specifically write an e.g. SHIFT-JIS-to-UTF-8 or UTF-EBCDIC-to-Big5-HKSCS pairwise function, it Just Worked™ as long as you had some common pivot between functions. But, it involved a fair amount of template shenanigans and a bunch of things that, quite frankly, do not exist in C.

Can we do the same in C, nicely?

Well, dear reader, it's time we found out! Is it possible to make an interface that can do the same as C++, in C? Can we express the ability to go from one arbitrary encoding to another arbitrary encoding? And, because we're talking about C, can it be made to be **fast**?

- simduff¹
- iconv² (patched vcpkg version, search “libiconv”)
- boost.text³ (proposed for Boost, in limbo…?)
- utf8cpp⁴
- ICU⁵ (plain libicu & friends, and presumably not the newly released ICU4X, or ICU4C/ICU4J)
- encoding_rs⁶ through its encoding_c⁷ binding library

¹<https://github.com/simduff/simduff>

²<https://www.gnu.org/software/libiconv/>

³<https://github.com/tzlaaine/text>

⁴<https://github.com/nemtrif/utfcpp>

⁵<https://unicode-org.github.io/icu/userguide/conversion/converters.html>

⁶https://github.com/hsivonen/encoding_rs

⁷https://github.com/hsivonen/encoding_c

- the Windows API, both `MultiByteToWideChar`⁸ and `WideCharToMultiByte`⁹
- ztd.text¹⁰, the initial C++ version of this library leading P1629¹¹

We will not only be comparing API design/behavior, but also the speed with benchmarks to show whether or not the usage of the design we come up with will be workable. After all, performance is part of correctness measurements. No use running an algorithm that's perfect if it will only compute the answer by the Heat Death of the Universe, right? So, with all of that in mind, let's start by trying to craft a C API that can cover all of the concerns we need to cover. In order to do that without making a bunch of mistakes or repeated bad ideas from the last five decades, we'll be looking at all the above mentioned libraries and how they handle things.

83.1 Enumerating the Needs

At the outset, this is a pretty high-level view of what we are going for:

1. know how much data was read from an input string, even in the case of failure;
2. know how much data was written to an output, even in the case of failure;
3. get an indicative classification of the error that occurred; and,
4. control the behavior of what happens with the input/output that happens when a source does not have proper data within it.

There will also be sub-concerns that fit into the above but are noteworthy enough to call out on their own:

1. the code can be/will be fast;
2. the code can handle all valid input values; and,
3. the code is useful for other higher level operations that are not strictly about encoding/decoding/transcoding stuff (validation, counting, etc.).

With all this in mind, we can start evaluating APIs. To help us, we'll create the skeleton of a table we're going to use:

⁸<https://docs.microsoft.com/en-us/windows/win32/api/stringapiset/nf-stringapiset-multibytetowidechar>

⁹<https://docs.microsoft.com/en-us/windows/win32/api/stringapiset/nf-stringapiset-widechartomultibyte>

¹⁰<https://github.com/soasis/text/>

¹¹https://thephd.dev/_vendor/future_cxx/papers/d1629.html

Feature Set ↓ vs. Library →	ICU	libiconv	simdutf	encoding_rs/en	ztd.text
Handles Legacy Encodings	?	?	?	?	?
Handles UTF Encodings	?	?	?	?	?
Bounded and Safe Conversion API	?	?	?	?	?
Assumed Valid Conversion API	?	?	?	?	?
Unbounded Conversion API	?	?	?	?	?
Counting API	?	?	?	?	?
Validation API	?	?	?	?	?
Extensible to (Runtime) User Encodings	?	?	?	?	?
Bulk Conversions	?	?	?	?	?
Single Conversions	?	?	?	?	?
Custom Error Handling	?	?	?	?	?
Updates Input Range (How Much Read TM)	?	?	?	?	?
Updates Output Range (How Much Written TM)	?	?	?	?	?

Feature Set ↓ vs. Library →	boost.text	utf8cpp	Standard C	Standard C++	Windows API
Handles Legacy Encodings	?	?	?	?	?
Handles UTF Encodings	?	?	?	?	?
Bounded and Safe Conversion API	?	?	?	?	?
Assumed Valid Conversion API	?	?	?	?	?
Unbounded Conversion API	?	?	?	?	?
Counting API	?	?	?	?	?
Validation API	?	?	?	?	?
Extensible to (Runtime) User Encodings	?	?	?	?	?
Bulk Conversions	?	?	?	?	?
Single Conversions	?	?	?	?	?
Custom Error Handling	?	?	?	?	?
Updates Input Range (How Much Read™)	?	?	?	?	?
Updates Output Range (How Much Written™)	?	?	?	?	?

The “?” just means we haven’t evaluated it / do not know what we’ll get. Before evaluating the performance, we’re going to go through every library listed here (for the most part; one of the libraries is mine (ztd.text) so I’m just going to brush over it briefly since I already wrote a big blog post about it and thoroughly documented all of its components) and talk about the salient points of each library’s API design. There will be praise for certain parts and criticism for others. Some of these APIs will be old; not that it matters, because many are still in use and considered fundamental. Let’s talk about what the feature sets mean:

83.1.1 Handles Legacy Encodings

This is a pretty obvious feature: whether or not you can process at least some (not all) legacy encodings. Typical legacy encodings are things like Latin-1, EUC-KR, Big5-HKSCS, Shift-JIS, and similar. Usually this comes down to a library trying (and failing) to handle things, or just not bothering with them at all and refusing to provide any structure for such legacy encodings.

83.1.2 Handles UTF Encodings

What it says on the tin: the library can convert UTF-8, UTF-16, and/or UTF-32, generally between one another but sometimes to outside encodings. Nominally, you would believe this is table-stakes to even be discussed here but believe it or not some people believe that not all Unicode conversions be fully supported, so it has to become a row in our feature table. We do not count specifically converting to UTF-16 Little Endian, or UTF-32 Big Endian, or what have you: this can be accomplished by doing an UTF-N conversion and then immediately doing byte swaps on the output code units of the conversion.

83.1.3 Safe Conversion API

Safe conversion APIs are evaluated on their ability to have a well-bounded input range and a well-bounded output range that will not start writing or reading off into space when used. This includes things like having a size go with your output pointer (or a “limit” pointer to know where to stop), and avoiding the use of C-style strings on input (which is bad because it limits what can be put into the function before it is forced to stop due to null termination semantics). Note that all encodings have encodings for null termination, and that stopping on null terminators was so pervasive and so terrible that it spawned an entire derivative-UTF-8 encoding so that normal C-style string operations worked on it (see Java Programming Language documentation, §4.4.7, Table 4.5¹² and other documentation concerning “Modified UTF-8”).

83.1.4 Assumed Valid Conversion API

Assumed valid conversion APIs are conversions that assume the input is already valid. This can drastically increased speed because checking for e.g. overlong sequences, illegal sequences, and other things can be entirely ignored. Note that it does not imply **unbounded** conversion, which is talked about just below. Assumed valid conversions are where the input is assumed valid; unbounded conversions are where the output is assumed to be appropriately sized/aligned/etc. Both are dangerous and unsafe and may lead to undefined behavior (unpredictable branching in algorithm, uncontrolled reads and stray writes, etc.) or buffer overruns. This does not mean it is always bad to have: Rust saw significant performance increases when they stopped verifying known-valid UTF-8 string data when constructing and moving around their strings, for both compile and run time workloads.

Lacking this API can result in speed drops, but not always.

83.1.5 Unbounded Conversion API

Unbounded conversions are effectively conversions with bounds checking on the output turned off. This is a frequent mainstay of not only old, crusty C functions but somehow managed to stay as a pillar of the C++ standard library. Unbounded conversions typically only take a single output iterator / output pointer, and generally have no built-in check to see if the output is exhausted for writing. This can lead to buffer overruns for folks who do not appropriately check or size their outputs. Occasionally, speed gains do come from unbounded writing but it is often more powerful and performance-impacting to have assumed valid conversion APIs instead. Combining both assumed valid and unbounded conversions tend to offer the highest performance of all APIs, pointing to a need for these APIs to exist even if they are not inherently safe.

¹²<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.4.7>

83.1.6 Counting API

This is not too much of a big deal for APIs; nominally, it just allows someone to count how many bytes / code units / code points result from converting an input sequence from X to Y, without actually doing the conversion and/or without actually writing to an output buffer. There are ways to repurpose normal single/bulk conversions to achieve this API without requiring a library author to write it, but for this feature we will consider the explicit existence of such counting APIs directly because they can be optimized on their own with better performance characteristics if they are not simply wrappers around bulk/single conversion APIs.

83.1.7 Validation API

This is identical to counting APIs but for checking validity. Just like counting APIs, significant speed gains can be achieved by taking advantage of the lack of the need to count, write, or report/handle errors in any tangible or detailed fashion. This is mostly for the purposes of optimization, but may come in handy for speeding up general conversions where checking validity before doing an assumed-valid conversion may be faster, especially for larger blocks of data.

83.1.8 Extensible to (Runtime) User Encodings

This feature is for the ability to add encodings into the API. For example, if the API does not have functions for TSCII, or recognize TSCII, is it possible to slot that into an API without needing to abandon what principles or goals the library sets up for itself? There is also the question of whether or not the extensibility happens at the pre-build step (adding extra data files, adding new flags, and more before doing a rebuild) or if it is actually accommodated by the API of the library itself.

83.1.9 Bulk Conversions

Bulk conversions are a way of converting as much input as possible to fill up as much output as possible. The only stopping conditions for bulk conversions are exhausted input (success), not enough room in the output, an illegal input sequence, or an incomplete sequence (but only at the very end of the input when the input is exhausted). Bulk conversions open the door to using Single Instruction Multiple Data (SIMD) CPU instructions, GPU processing, parallel processing, and more to convert large regions of text at a time.

More notably, given a stable single conversion function, running that conversion in a loop would produce the same effect, but may be slower due to various reasons (less able to optimize the loop, cannot be easily restructured to use SIMD, and more). Bulk conversions get around that by stating up-front they will process as much data as possible.

83.1.10 Single Conversions

Single conversions are, effectively, doing “one unit of work” at a time. Whether converting Unicode one code point at a time or making iterators/views to go over a range of UTF-8 one bundle of non-error code units at a time, the implication here is that it uses minimal memory while guaranteeing that forward progress is made/work is done. It is absolutely not the most performant way of encoding, but it makes all

other operations possible out of the composition of this single unit of work, so there's that. If a single conversion is the only thing you have, you can typically build up the bulk conversion from it.

In the opposite case, where only a bulk conversion API is available, you can still implement a single conversion API. Just take a bulk API, break the input off into a subrange of size 1 from the beginning of the input. Then, call the bulk API. If it succeeds, that's all you need to do. If not, you take the subrange and make it a subrange of size 2 from the start of the input. You keep looping up until the bulk API successfully converts that sub-chunk of input, or until input exhaustion. This method is, of course, horrifically inefficient. It is inadvisable to do this, unless correctness and feature set is literally your only goal with your library. Pressuring your user to provide a single conversion first, and then a bulk conversion, will provide far better performance metrics.

83.1.11 Custom Error Handling

This is just the ability for an user to change what happens on failure/error. Note that this is still possible if the algorithm just stops where the error is and hands you that information; you can decide to skip and/or insert your own kind of replacement characters. (Or panic/crash, write to log, trigger an event; whatever it is you like to do.) I did not think I needed this category, but after working with a bunch of different APIs it is surprising how many do not offer functions to handle this exact use case and instead force replacements or just throw this information into the sea of forgetfulness.

83.1.12 Updates Input Range / Updates Output Range

It's split into two categories so that I can document which part of this that APIs handle appropriately. This is for when, on either success or error/failure, an API tells you where in the input it failed (the "input range" bit) if it did, and how much it wrote before it succeeded/failed (the "output range" bit). I was always annoyed by this part, and I only got increasingly annoyed that it seems most APIs just give you a big fat middle finger and tell you to go pound sand when it comes to figuring out what happened with the data. This also ties in with "custom error handling" quite a bit; not returning this information means that even if the user was prepared to do it on their own, it is fundamentally impossible to since they lose any possible information of where errors occurred or how much output was written out to not re-do work. Oftentimes, not getting this information results in you needing to effectively treat the entire buffer of input and the entire buffer of output as one big blob of No-Man's-Land. You could have 4 GB of input data resulting in 8.6 GB of output data, and other APIs will literally successfully convert all but the very last byte and then simply report "we messed up, boss".

Where did you mess up?

"(:)"

Okay, but how far did you get before you messed up?

"(:):)"

Okay, okay, easy there, how much did you actually manage to get out before you think you messed up?

"(:)\u2225(\u0333)\u2225(:)"

D---D-Do you remember ANYTHING about what we just spent several minutes of compute...doing...?
?

"Oh, hey boss! You ready to get started on all this data? It's gonna be great."

...

If that conversation is concerning and/or frustrating to you, then welcome to the world of string and text APIs in C and C++. It's having this conversation, every damn day, day in and day out, dusk to dawn. It's pretty bad!

83.1.13 Nevertheless

Now that we have all of our feature sets and know what we are looking for in an API, we can start talking about all of these libraries. The core goal here will be in dealing with the issues of each API and trying to fill out as much of the functionality as required in the tables. The goal will be to fill it all with **yes** in each row, indicating full support. If there is only partial support, we will indicate that with :) and add notes where necessary. If there is no support, we will use a **no**.

83.2 ICU

ICU has almost everything about their APIs correct, which is great because it means we can start with an almost-perfect example of how things work out. They have a number of APIs specialized for UTF-8 and UTF-16 conversions (and we do benchmark those), but what we will evaluate here is the `ucnv_convertEx` API that serves at their basic and fundamental conversion primitive. Here's what it looks like:

```
U_CAPI void ucnv_convertEx(
    UConverter *targetCnv, UConverter *sourceCnv, // converters describing the encodings
    char **target, const char *targetLimit, // destination
    const char **source, const char *sourceLimit, // source data
    UChar *pivotStart, UChar **pivotSource,
    UChar **pivotTarget, const UChar *pivotLimit, // pivot
    UBool reset, UBool flush, UErrorCode *pErrorCode); // error code out-parameter
```

Some things done really well are the `pErrorCode` value and the pointer-based source/limit arguments. The error code pointer means that an user cannot ever forget to pass in an error code object. And, what's more, is that if the value going into the function isn't set to the proper 0-value (`U_SUCCESS`), the function will return a "warning" error code that the value going in was an unexpected value. This forces you to initialize the error code with `UErrorCode error = U_SUCCESS;`, then pass it in properly to be changed to something. You can still ignore what it sets the value to and pretend nothing ever happened, but you have, at least, forced the user to reckon with its existence. If someone is a Space Cadet like me and still forgets to check it, well. That is on them.

Furthermore, the pointer arguments are also extraordinarily helpful. Taking a pointer-to-pointer of the first argument means that the algorithm can increment the pointer past what it has read, to successfully indicate how much data was read that produced an output. For example, UTF-8 may have anywhere from 1 to 4 code units of data (1 to 4 `unsigned chars` of data) in it. It is impossible to know how much data was read from a "successful" decoding operation, because it is a variable amount. Due to this, you cannot know how much was read/written without the function telling you, or you going backwards to re-do the work the function already did.

Updating the pointer value makes sure you know how much input was successfully read. An even better version of this updates the pointer only if the operation was successful, rather than just doing it willy-nilly.

This is the way ICU works, and it is incredibly helpful when `*pErrorCode` is not set to `U_SUCCESS`. It allows you to insert replacement characters and/or skip over the problematic input if needed, allowing for greater control over how errors are handled.

Of course, there's some things the ICU function is not perfectly good at, and that is having potentially unchecked functions for increased speed. One of the things I attempted to do while using `ucnv_convertEx` was pass in a `nullptr` for the `targetLimit` argument. The thinking was that, even if that was not a valid ending pointer, the `nullptr` would never compare equal to any valid pointer passed through the `*target` argument. Therefore, it could be used as a pseudo-cheap way to have an “unbounded” write, and perhaps the optimizer could take advantage of this information for statically-built versions of ICU. Unfortunately, if you pass in `nullptr`, ICU will immediately reject the function call with `U_ILLEGAL_ARGUMENT` as the error code. You also cannot do “counting” in a straightforward manner, since `nullptr` is not allowed as an argument to the `target`-parameters.

There are other dedicated functions you can use to do counting (“pre-flighting”, as the ICU documentation calls it since it also performs other functions), and there are also other conversion functions you can do to instrument the “in-between” part of the conversion too (that is serviced by the `pivotStart`, `pivotSource`, `pivotTarget`, and `pivotLimit` parameters). But, all in all it still contains much of the right desired shape of an API that is useful for text encoding, and does provide a way to do conversions between arbitrary encodings.

Single conversion or one-by-one conversions are supported by using the `CharacterIterator` abstraction, but it has very strange usability semantics because it iterates over `UChars` of 16 bits for UTF-16. Stitching together surrogate characters takes work and, in most cases, is almost certainly not worth it given the API and its design for this portion.

For going to UTF-8 or UTF-16—which should cover the majority of modern conversions—there are also dedicated APIs for it, such as `ucnv_fromUChars/ucnv_toUChars` and `u_strFromUTF8` and `u_strToUTF8`. These APIs take different parameters but share much of the same philosophies as the `ucnv_convertEx` described above, generally lacking a `pivot` buffer set of arguments since direct Unicode conversions need no in-between pivot. Unlike many of the other libraries compared / tested in this API and benchmark comparison we are doing, they do support quite an array of legacy encodings. The only problem is that adding new encoding conversions takes rebuilding the library and/or modifying data files to change what is available.

Despite being an early contender and having to base their API model around Unicode 1.0 and a 16-bit `UChar` (and thus settling on UTF-16 as the typical go-between), ICU’s interface has at least made amends for this by providing a rich set of functionality. It can be difficult to figure it all out and how to use it appropriately, but once you do it works well. It may fail in some regards due to not embracing different kinds of performant API surfaces but by-and-large it provides everything someone could need (and more, but we are only concerned with encoding conversions at this point).

83.3 simdutf

As its name states, `simdutf` wants to be a Single Instruction Multiple Data implementation (SIMD) for all of the Unicode Transformation Formats (UTFs). And it does achieve that; with speeds that tear apart tons of data and rebuild it in the appropriate UTF encoding, Professor Lemire earned himself a sweet spot in the SPIRE 2021: String Processing and Information Retrieval Journal/Symposium thingy for his paper,

“Unicode at Gigabytes per Second”¹³. The interface is markedly simpler than ICU’s; with no generic conversions to worry about and no special conversion features, simdutf takes the much simpler route of doing the following:

```
simdutf_warn_unused size_t convert_utf8_to_utf16le(
    const char * input,
    size_t length,
    char16_t* utf16_output) noexcept;
simdutf_warn_unused result convert_utf8_to_utf16le_with_errors(
    const char * input,
    size_t length,
    char16_t* utf16_output) noexcept;
simdutf_warn_unused size_t utf8_length_from_utf16le(
    const char16_t* input,
    size_t length) noexcept;
simdutf_warn_unused bool validate_utf8(
    const char *buf,
    size_t len) noexcept;
simdutf_warn_unused result validate_utf8_with_errors(
    const char *buf,
    size_t len) noexcept;
```

`simdutf_warn_unused` is a macro expanding to `[[nodiscard]]` or similar shenanigans like `__attribute__((warn_unused))` on the appropriate compiler(s). The `result` type is:

```
struct result {
    error_code error;
    size_t position;
};
```

It rinses and repeats the above for UTF-8 (using `(const)char*`), UTF-16 (using `(const)char16_t*`), and UTF-32 (using `(const)char32_t*`). You will notice a couple of things lacking from this interface, before we get into the “SIMD” part of simdutf:

1. How much input have I read, if something goes wrong?
2. How much output have you written, if something goes wrong?
3. How do I know if the buffer `utf8/16/32_output` is full?

For the version that is not suffixed with `_with_errors`, the first two cases are assumed not to happen: there are no output errors because it assumes the input data is well-formed UTF-N (for N in 8, 16, 32). This only returns a `size_t`, telling you how much data was written into the `*_output` pointer. The assumption is that the entire input buffer was well-formed, after all, and that means the entire input buffer was consumed, assuming no problems. The `*_with_errors` functions have just one problem, however…

¹³<https://arxiv.org/abs/2111.08692>

83.3.1 One, The Other, But Definitely Not Both.

You are a smart, intelligent programmer. You know data coming in can have the wrong values frequently, due to either user error, corruption, or just straight up maliciousness. Out of an abundance of caution, you allocate a buffer that is big enough. You run the `convert_utf8_to_utf16le_with_errors` function on the input data, not one to let others get illegal data past you! And you were right to: some bad data came in, and the `result` structure's `error` field has an `enum error_code` of `OVERLONG`: hah! Someone tried to sneak an overlong-UTF-8-encoded character into your data, to trap you! You pat yourself on the back, having caught the problem. But, now... hm! Well, this is interesting. You have both an `input` pointer, and an `utf16_output` pointer, but there's only one `size_t position` field! Reading the documentation, that applies to the input, so... okay! You know where the badly-encoded overlong UTF-8 sequence is! But...uhm. Er.

How much output did you write again...?

This is simdutf's problem. If you do a successful read of all the input, and output all the appropriate data, the `position` field on the `result` structure tells you how many characters were written to the output. You know it was successful, so you know you've consumed all the input; that's fine! But when an errors occurs? You only know how much input was processed before the error. Did you write 8.6 GB of data and only failed on the very last byte? Did you want to not start from the beginning of that 8.6 GB buffer and page in a shitload of memory? Eat dirt, loser; we're not going to tell you squat. Normally, I'd be a-okay with that order of business. But there's just one teensy, tiny problem with simdutf here! If you go into the implementations (the VARIOUS implementations, using everything from SSE2 to AVX2), you'll notice a particular...pattern:

```
.cmake > build > _deps > simdutf-src > src > haswell > avx2_convert_utf8_to_utf16.cpp > convert_masked_utf8_to_utf16
 7 // It returns how many bytes were consumed (up to 12).
 8 size_t convert_masked_utf8_to_utf16(const char *input,
 9     uint64_t utf8_end_of_code_point_mask,
10     char16_t *utf16_output) {
11     // we use an approach where we try to process up to 12 input bytes.
12     // Why 12 input bytes and not 16? Because we are concerned with the size of
13     // the lookup tables. Also 12 is nicely divisible by two and three.
14     //
15     //
16     // Optimization note: our main path below is load-latency-dependent. Thus it
17     // is beneficial to have fast paths that depend on branch prediction but have low
18     // This results in more instructions but, potentially, also higher speeds.
19     //
20     // We first try a few fast paths.
21     const __m128i in = _mm_loadu_si128((__m128i *)input);
22     const uint16_t input_utf8_end_of_code_point_mask =
23         utf8_end_of_code_point_mask & 0xffff;
24     if(((utf8_end_of_code_point_mask & 0xffff) == 0xffff)) {
25         // We process the data in chunks of 16 bytes.
26         _mm256_storeu_si256(reinterpret_cast<__m256i *>(utf16_output), _mm256_cvtepi32_ps(
27             utf16_output += 16; // We wrote 16 16-bit characters.
28         return 16; // We consumed 16 bytes.
29     }
30     if(((utf8_end_of_code_point_mask & 0xffff) == 0xaaaa)) {
```

It. Knows.

It knows how much it's been writing, and it just deigns not to tell you when its done. And I can see why it doesn't pass that information back. After all, we all know how **expensive** it is to have an extra

`output_position` field. That's a **WHOLE** wasted `size_t` in the case where we read the input successfully and output everything nicely; it would be silly to include it! If we did not successfully read everything, what good can the input be anyhow?

Sarcasm aside, simdutf gets so many points for having routines that assume validity and do not, as well as length counting functions for input sequences and more, but just drops the ball on this last crucial bit of information! You either get to know the input is fully consumed and the output you wrote, or where you messed up in the input sequence, but you can't get both.

Of course, it also doesn't take buffer safety seriously either. Not that I blame simdutf for this: this has been an ongoing problem that C and C++ continue to not take seriously in all of its APIs, new and old. Nothing exemplifies this more than the standard C and C++ ways of "handling" this problem.

83.3.2 A Brief & Ranting Segue: Output Ranges and "Modern" C++

Since the dawn of time, C and C++ have been on team "output limits are for losers". I wrote an extensive blogpost about Output Ranges and their benefits¹⁴ after doing some benchmarks and citing Victor Zverovich's work on fmtlib¹⁵. At the time, that blogpost was fueled by rumblings of the idea that we do not need output ranges, or even a single output iterator (which is like an output pointer `char16_t* utf16_output` that Lemire's functions take). Instead, we should take sink functions, and those sink functions can just be optimized into the speedy direct writes we need. The blog post showed that you can not only have the "sink" based API thanks to Stepanov's iterator categorizations (an output iterator does exactly what a "sink" function is meant to do), but you can also get the performance upgrades to a direct write by having an output range composed of contiguous iterators of $[T^*, T^*)/[T^*, size_t]$. This makes output ranges both better performing and, in many cases safer than both single-iterator and sink functions. So, when we standardized ranges, what did we do with old C++ functions that had signatures like

```
namespace std {
    template <typename InputIt, typename OutputIt>
    constexpr OutputIt copy(
        InputIt first,
        InputIt last,
        OutputIt d_first);
}
```

the above? Well, we did a little outfitting and made the ones in `std::ranges` look like...

```
namespace std {
    namespace ranges {
        template <std::input_iterator I, std::sentinel_for<I> S, std::weakly_incrementable O>
        requires std::indirectly_copyable<I, O>
        constexpr copy_result<I, O> copy(
            I first,
            S last,
            O result); // ... well, shit.
```

¹⁴<https://thephd.dev/output-ranges#t-t>

¹⁵<https://github.com/fmtlib/fmt>

```

    }
}

```

...Oh. We...we still only take an output iterator. There's no range here. Well, hold on, there's a version that takes a range! Surely, in the version that takes an input range, O will be a range too-

```

namespace std {
    namespace ranges {
        template <ranges::input_range R, std::weakly_incrementable O>
        requires std::indirectly_copyable<ranges::iterator_t<R>, O>
        constexpr copy_result<ranges::borrowed_iterator_t<R>, O> copy(
            R&& r, // yay, a range!
            O result); // ... lmao
    }
}

```

Ah.

...Nice. Nice. Nice nice cool cool. Good. Great, even.

Fantastic.

One of the hilarious bits about this is that one of the penalties of doing SIMD-style writes and reads outside the bounds of a proper data pointer can be corruption and/or bricking of your device. If you have a pointer type for O result, and you start trying to do SIMD or other nonsense without knowing the size (or having an end pointer which can be converted into a size) with the single output pointer, on some hardware going past the ending boundary of the data and working on it means that you can, effectively, brick the device you're running code on.

Now, this might not mean anything for std::(ranges::)copy, which might not rely on SIMD at all to get its work done. But, this does affect all the implementations that do want to use SIMD under the hood and may need to port to more exotic architectures; not having the size means you can't be sure if/when you might do an "over-read" or "over-write" of a section of data, and therefore you must be extra pessimistic when optimizing on those devices. To be clear: a lot of computing does not run on such devices (e.g., all the devices Windows runs on and cares about do not have this problem). But, if you're going to be writing a standard it might behoove us to actually give people the tools they need to not accidentally destroy their own esoteric devices when they use their SIMD instructions. When you have a range (in particular, a contiguous range with a size), you can safely work within the boundaries of both the input and output data and not trigger spurious failures/device bricking from being too "optimistic" with reads and writes outside of boundaries.

The weird part is that we also already have a range-based solution to "if I have to take a range, then I'm forced to bounds check against that range". If you take an output range, you can also take an infinity range that simply does unbounded writes. This is something I've been using extensively since the earliest range-v3 days: [unbounded_view](#). In fact, I gave a whole talk about how by using output ranges you can get safety by-default (the right default) and then get speed when you want it in an easily-searchable, greppable manner (timed video link¹⁶):

It still baffles me that we can't push people with our standard APIs to have decent performance metrics with safety first, and then ask people to deliberately pull the jar lid off to get to the dangerous and terrifying

¹⁶<https://youtu.be/BdUipluIfIE?t=2267>

JeanHeyd Meneide

Catch ↑: The (Baseline) Unicode for C++23

Need speed?

- Ask for it with an unbounded view

```
std::u8string input = u8"ΜΑΓ ΓΑΕΣ ΤΙΑΝ, ΝΙ ΜΙΣ ΥΝ ΗΣΑΝ ΒΡΙΤΤΙΨ。";
std::u32string output(16, U'\0');
std::text::utf8 encoding{};
std::text::utf8::state state{};

auto result = encoding.decode(
    input, std::ranges::unbounded_view(output.data()),
    state, std::text::default_handler{}
); // does not overrun buffer
```

C++ mojo later. But, we continue to do this for C, C++, and similar code without taking a whole-library or whole-standard approach to being conscientious of the security vulnerability-rich environments we like to saddle developers with. These sorts of decisions are infectious because they are effectively the standards-endorsed interfaces, and routinely we suffer from logic errors which leak into unchecked buffer overruns and worse because the every-day tools employed in C and C++ codebases next to the usual logic we do are often the most unsafe possible tools we have. You cannot debug build or iterator-checking your way out of fundamentally poor design decisions. No matter how hard Stephan T. Lavavej or Louis Dionne or Jonathan Wakely iterator-safety the standard libraries in debug mode, leaving open the potential for gaping issues in our release builds is not helpful for the forward progress of C and C++ to be considered effective languages for an industry suffering from a wide variety of increasingly sophisticated security attacks.

But I digress.

The real problem here is that, in simdutf, if your data is not perfectly valid you are **liable to waste work done in the face of a failed conversion**. Kiss that 8.5999 GB goodbye and prepare to start from the beginning of that buffer all over again, because the interface still does not return how much output was written! In at least one win for Modern C++ interfaces, the new `std::ranges` algorithms in C++ did learn from the past at least a little bit. They return both the input iterator and the output iterator (`std::ranges::copy_result<I, O>`) passed into the function. simdutf has, unfortunately, been learning from the old C++ and C school of functions rather than the latest C++ school of functions! So, even if they both make the same unbounded-output mistake, simdutf doesn't get the updated input/output perspective correct. And I really do mean the C school of function calls: the Linux Kernel has gotten into this same situation with trying to make a string copy function!

The Kernel folks are now deprecating `strlcpy`. They have begun the (long, painful?) maybe-migration to the newly decided-on `strncpy`. They are, once again, trying to convince everyone that the new `strncpy` is better than `strlcpy`, the same way the people who wrote `strlcpy` convinced everyone that it was better than `strncpy`. This time, they declare, they have really cracked the string copy functionality and came up with the optimal return values and output processing. And you know what, maybe for a bulk of the situations they care about, the people who designed `strncpy` are right! Unfortunately, you get tired of reading about the same return-value-but-changed-slightly or null-termination-but-we-handled-it-better-this-time-I-swear mistakes over and over again, you know? Even the article writer is resigned to this apparent infinity-cycle of “let’s write a

new copy function” every decade or two (emphasis mine):

…That would end a 20-year sporadic discussion on the best way to do bounded string copies in the kernel —all of those remaining `strncpy()` calls notwithstanding —at least until some clever developer comes up an even better function and starts the whole process anew.

Jonathan Corbet, August 25th, 2022¹⁷

I wish we would give everyone the input-and-output bounded copies with a full set of error codes so they could capture all the situations that matter, and then introduce `strlcpy`/`strncpy`/`strscpy` as optimizations where they are confident they can introduce it as such. But, instead, we’re just going to keep subtly tweaking/modifying/poking at the same damn function every 20 years. And keep introducing weird, messed up behavioral intrigues that drive people up the wall¹⁸. It does give us all something to do, I guess! Clearly, we do not have enough things to be working on at the lowest levels of computing, beneath all else, except whether or not we’ve got our string copy functions correct. That’s the kind of stuff we need to be spending the time of the literal smartest people on the earth figuring out. Again. And get it right this time! For real. We promise. Double heart-cross and mega hope-to-die promise. Like SUPER-UBER pinky promise, it’s perfect this time, ultra swearries!!

Ultra swearries…

83.3.3 Rant Over

Regardless of how poorly output pointers are handled in the majority of C and C++ APIs, and ignoring the vast track record of people messing up null termination, sizes, and other such things, simdutf has more or less a standard interface offering **most** of the functionality you could want for Unicode conversions. Its combination of functions that do not check for valid input and ones that do (which are suffixed with `_with_errors`) allows for getting all of the information you need, albeit sometimes you need to make multiple function calls and walk over the data multiple times (e.g., call `validate_utf8` before calling `utf8_length_from_utf16le` since the length function does not bother doing validation).

simdutf also does not bother itself with genericity or pivots or anything, because it solely works for a fixed set of encodings. This makes it interesting for Unicode cases (which is hopefully the vast majority of encoding conversions performed), but utterly useless when someone has to go battle the legacy dragons that lurk in the older codebases.

83.4 utf8cpp

utf8cpp is what it says on the tin: UTF-8 conversions. It also includes some UTF-16 and UTF-32 conversion routines and, as normal for most newer APIs, does not bother with anything else. It has both checked and unchecked conversions, and the APIs all follow an STL-like approach to storing information.

```
template <typename u16bit_iterator, typename octet_iterator>
u16bit_iterator utf8to16 (
    octet_iterator start,
    octet_iterator end,
```

¹⁷<https://lwn.net/Articles/905777/>

¹⁸<https://twitter.com/saleemrashid/status/1406661700900823051>

```

        u16bit_iterator result);

namespace unchecked {
    template <typename u16bit_iterator, typename octet_iterator>
    u16bit_iterator utf8to16 (
        octet_iterator start,
        octet_iterator end,
        u16bit_iterator result);
}

```

You can copy-and-paste all of my criticisms for simdutf onto this one, as well as all my praises. Short, simple, sweet API, has an explicit opt-in for unchecked behavior (using a `namespace` to do this is a nice flex), and makes it clear what is on offer. As a side benefit, it also includes an `utf8::iterator` and an `utf16::iterator` classes to do iterator and view-like stuff with, which can help cover a pretty vast set of functionality that the basic functions cannot provide.

It goes without saying that extensibility is not built into this package, but it will be fun to test its speed. The way errors are handled are done by the user, which means that custom error handling / replacement / etc. can be done. Of course, just like simdutf, it thinks input iterator returns are for losers, so there's no telling where exactly the error might be in e.g. an UTF-16 sequence or something to that effect. However, for ease-of-use, utf8cpp also includes a `utf8::replace_invalid` function to replace bad UTF-8 sequences with a replacement character sequence. It also has `utf8::find_valid`, so you can scan for bad things in-advance and either remove/replace/eliminate them in a given object yourself. (UTF-8 only, of course!)

83.5 encoding_rs/encoding_c

`encoding_rs` is, perhaps surprisingly, THE Rust entry point into this discussion. This will make it interesting from a performance perspective and an API perspective, since it has a C version —`encoding_c`— that provides a C-like API with a C++ wrapper around it where possible. It's got a much weirder design philosophy than freely-creatable conversion objects; it uses static const objects of specific types to signal which encoding is which:

```

// ...

/// The UTF-8 encoding.
extern ENCODING_RS_NOT_NULL_CONST_ENCODING_PTR const UTF_8_ENCODING;

/// The gb18030 encoding.
extern ENCODING_RS_NOT_NULL_CONST_ENCODING_PTR const GB18030_ENCODING;

/// The macintosh encoding.
extern ENCODING_RS_NOT_NULL_CONST_ENCODING_PTR const MACINTOSH_ENCODING;

// ...

```

The types underlying them are all the same, so you select whichever encoding you need either by referencing the static const object in code or by using the `encoding_for_label(uint8_t const* label, s_`

`ize_t` `label_len`) function. You then start calling the `(decoder|encoder)_decode|encode_(to|from)_utf16` functions (or using the object-oriented function calls that do exactly the same thing but by calling `(decoder|encoder)->(decode|encode)_to|from_utf16` on a `decoder/encoder` pointer):

```
uint32_t encoder_encode_from_utf16(
    ENCODING_RS_ENCODER* encoder,
    char16_t const* src,
    size_t* src_len,
    uint8_t* dst,
    size_t* dst_len,
    bool last,
    bool* had_replacements);

uint32_t encoder_encode_from_utf16_without_replacement(
    ENCODING_RS_ENCODER* encoder,
    char16_t const* src,
    size_t* src_len,
    uint8_t* dst,
    size_t* dst_len,
    bool last);

uint32_t decoder_decode_to_utf16(
    ENCODING_RS_DECODER* decoder,
    uint8_t const* src,
    size_t* src_len,
    char16_t* dst,
    size_t* dst_len,
    bool last,
    bool* had_replacements);

uint32_t decoder_decode_to_utf16_without_replacement(
    ENCODING_RS_DECODER* decoder,
    uint8_t const* src,
    size_t* src_len,
    char16_t* dst,
    size_t* dst_len,
    bool last);
```

First off, I would just like to state, for the record:

FINALLY. Someone finally included some damn sizes to go with both pointers. This is the first API since ICU not to just blindly follow in the footsteps of either the standard library, C string functions, or whatever other nonsense keeps people from writing properly checked functions (with optional opt-outs for speed purposes). This is most likely due to the fact that this is a Rust library underneath, and the way data is handled is with built-in language slices (the equivalent of C++'s `std::span`, and the equivalent of C's nothing because C still hates its users and wants them to make their own miserable structure type with horrible usability interfaces¹⁹). `encoding_rs` unfortunately fails to provide functions that do no checking

¹⁹https://ztdidk.readthedocs.io/en/latest/c_api/c_span.html

here. Weirdly enough, this functionality could be built in by allowing `dst_len` to be `NULL`, giving the “write indiscriminately into `dst` and I won’t care” functionality. But, `encoding_rs` just…does not, instead stating:

…UB ensues if any of the pointer arguments is `NULL`, `src` and `src_len` don’t designate a valid block of memory or `dst` and `dst_len` don’t designate a valid block of memory. …

So, that’s that. Remember that my qualm is not that there are unsafe versions of functions: it’s that there exist unsafe functions **without well-designed, safe alternatives**. `encoding_rs` swings all the way in the other direction, much like ICU, and says “unbounded writing is for losers”, leaving the use case out in the cold. The `size_t` pointer parameters still need to be as given, because the original Rust functions return sizes indicating how much was written. Rather than returning a structure (potentially painful to do in FFI contexts), these functions load up the new size values through the `size_t*` pointers, showing how much is left in the buffers.

Error handling can be done automatically for you by using the normal functions, with an indication that replacements occurred in the output `bool*` parameter `has_replacements`. Functions which want to apply some of their own handling and not just scrub directly over malformed sequences have to use the `_without_replacement`-suffixed functions.

Finally, the functions present here always go: to UTF-8 or UTF-16; or, from UTF-8 or UTF-16. It is your job to write a function that stiches 2 encodings together, if you need to go from one exotic/legacy encoding to another. This is provided in examples (here²⁰, and here²¹), but not in the base package: transcoding between any 2 encodings is something you must specifically work out. The design is also explicitly not made to be extensible; what the author does is effectively his own package-specific hacks to pry the mechanisms and Traits open with his bare hands to get the additional functionality (such as in this crate²²).

This makes it a little painful to add one’s own encodings using the library, but it can technically be done. I will not vouch for such a path because when the author tells me “I explicitly made it as hard as possible to make it extensible²³”, I don’t take that as an invitation to go trying to force extensibility. Needless to say, the API was built for streaming and is notable because it is used in Mozilla Firefox and a handful of other places like Thunderbird. It is also frequently talked up as THE Rust Conversion API, but that wording has only come from, in my experience, Mozilla and Mozilla-adjacent folks who had to use the Gecko API (and thus influenced by them), so that might just be me getting the echo feedback from a specific silo of Rustaceans. But if it’s powering things like Thunderbird, it’s got to be good, especially on the performance front, right?

I did encounter a pretty annoying usability bug when trying to convert from UTF-8 to UTF-16 the “generic” way, and ended up with an unexplained spurious conversion failure that seemed to mangle the data. It was, apparently, derived from the fact that you cannot ask for an encoder (an “output encoding”) of an UTF-16 type, which is itself apparently a restriction derived from the WHATWG encoding specification:

4.3. Output encodings

To get an output encoding from an encoding encoding, run these steps:

If encoding is replacement or UTF-16BE/LE, then return UTF-8. Return encoding —§4.3 WHATWG Encoding Specification, June 20, 2022²⁴

²⁰https://github.com/hsivonen/recode_cpp

²¹https://github.com/hsivonen/recode_c

²²<https://crates.io/crates/charset>

²³<https://twitter.com/hsivonen/status/1576068684300156928>

²⁴<https://encoding.spec.whatwg.org/#output-encodings>

Yes, you read that right. If the encoding is UTF-16, return the UTF-8 encoding instead. Don't raise an error, don't print that something's off to console, just slam an UTF-8 in there. I spent a good moment doing all sorts of checks/tests, trying to figure out why the Rust code was apparently giving me an UTF-8 encoder when I asked for an UTF-16 encoder:

```

is_utf16: false
UTF_16LE_ENCODING: 0x00007ff6e2fd45b0 {ztd::cuneicode::benchmarks::speed::execlencoding_c::ConstEncoding encoding_c_
  v ulenc: unique_ptr<encoding> 0x00007ff6e306eb68 {ztd::cuneicode::benchmarks::speed::execlencoding_rs::Encoding enco_
  v [ptr]: 0x000001b5f9e9f180 [encoding=0x00007ff6e306eb68 {ztd::cuneicode::benchmarks::speed::execlencoding_rs::Encoder_
  v encoding: 0x00007ff6e306eb68 {ztd::cuneicode::benchmarks::speed::execlencoding_rs::Encoding encoding_rs:::UTF_...
  > name: {data_ptr=0x00007ff6e306eb60 "UTF-8" length=5}
  > variant: {variant0={value=_0={table=0xcxxxxxxxxxxxxx {??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, -...
  > variant: {variant0={value=_0={table=0xcxxxxxxxxxxxxx {??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, ??, -...
  > [deleter]: default_delete
  v CALL STACK
  v Main Thread
  ztd::cuneicode::benchmarks::speed::execlencoding_c::convert_through_utf8(encoding_rs::Decoder * decoder, encoding_rs
PAUSED ON BREAKPOINT

```

The screenshot shows a debugger interface with assembly code at the top and C++ source code below. The assembly code includes labels like `is_utf16` and `UTF_16LE_ENCODING`. The source code is in a file named `execlencoding_c.cpp` with line numbers 51 to 66. Line 63 contains the problematic line: `if (encoder_result != INPUT_EMPTY) {`. The assembly code on line 63 corresponds to this line in the source code.

I'm certainly glad that `encoding_rs` cleaves that closely to the encoding specification, but you can backdoor this by simply generating a decoder for the encoding you are going from and directly calling `decoder->decode_to_utf16_without_replacement(...)`. This, of course, begs the question of why we are following the specification this closely in the first place, if I can simply cheat my way out of the situation by shooting the `encoder` portion in the face and doing a direct conversion from the `decoder` half. It also begs the question of why the WHATWG specification willingly returns you a false encoding rather than raising an error. I'm sure there's a good reason, but `encoding_rs` does not say it (other than stony-faced "it's what the spec does"), and the WHATWG spec does not make it immediately obvious what the benefit of this is supposed to be. So I will simply regard it as the infinite wisdom of people 1,000 times my superior and scold myself for being too dumb to read the docs appropriately.

Topping off my Unicode Conversion troubles, `encoding_rs` (and it's various derivatives like `charset`) don't believe in UTF-32 as an encoding either. To be fair, neither does the WHATWG specification, but I've got applications trafficking in UTF-32 text (including e.g. the very popular Harfbuzz shaper and the Freetype API), so...I guess we're just ignoring the hell out of those use cases and all of the `wchar_t`-based code out there in the world for *nix distributions.

Finally, there does not seem to be an "assume input is valid" conversion API either, despite the Rust ecosystem itself needing to have such functionality to drastically improve its own UTF-8 compile-time and run-time workloads for known-good strings. It's not the end of the world to have neither unbounded nor assumed valid conversions, but it certainly means that there could be plenty of performance left out on the table from the API here. We also have to remember that `encoding_rs`'s job is strictly for web code, and maybe they just don't trust anyone to do a conversion the unsafe way without endangering too-important data. Which is likely a fair call to make, but as somebody that's trying to crush Execution and Wide Execution encodings from the C libraries like a watermelon between my thighs the library comes up disappointingly short on necessary functionality.

It has certainly colored my impression of Rust's text encoding facilities if this is the end-all, be-all crate that was hyped up to me for handling things in Rust.

83.6 libiconv

`libiconv` has an interface similar to ICU, but entirely slimmed down. There is no pivot parameter, and instead of taking a pair of `[T**, T*)`, it works on -perhaps strangely `-[T**, size_t*)`:

```
size_t iconv(
    iconv_t cd,
    char ** inbuf,
    size_t * inbytesleft,
    char ** outbuf,
    size_t * outbytesleft);
```

Initially, when I first encountered this, I thought libiconv was doing something special here. I thought they were going to use the `nullptr` argument for `outbytesleft` and `outbuf` to add additional modes to the functions, such as:

1. input validation (`iconv(cd, inbuf, inbytesleft, nullptr, nullptr)`), similar to `validate_utf8` from `simdutf`;
2. output size counting (`iconv(cd, inbuf, inbytesleft, nullptr, outbytesleft)`), similar to `utf16le_length_from_utf8` from `simdutf`; and,
3. unbounded output writing (`iconv(cd, inbuf, inbytesleft, outbuf, nullptr)`), similar to the lack of an “end” or “size” done by `simdutf` and `utf8cpp` and many other APIs.

libiconv was, of course, happy to NOT provide any of that functionality, nor give other functions capable of doing so either. This was uniquely frustrating to me, because the shape of the API was ripe and ready to provide all of these capabilities, and provide additional functions for those capabilities. Remember up above when, about how a bulk API can be repurposed for the goals of doing counting, unbounded output writing, and validation? This is exactly what was meant: using the provided API surface of libiconv could achieve all of these goals as a bulk encoding provider. You could even repurpose the `iconv` API to do one-by-one encoding (the sticking point being, of course, that performance would be crap).

The only thing going for libiconv, instead, is its wide variety of encodings it supports. Other than that, it’s a decent API surface whose potential is not at all taken advantage of, including the fact that despite having a type-erased encoding interface does not provide any way to add new encodings to its type-erased interface. (If you want to do that, you need to add it manually into the code and then recompile the entire library, giving no means of runtime addition that are not expressly added to it by some outside force.)

Additionally, the names given to “create `iconv_t` conversion descriptor objects” function is not stable. For example, asking for the “UTF-32” encoding does not necessarily mean you will be provided with the UTF-32 encoding that matches the endianness of the machine compiled for. (This actually became a problem for me because a DLL meant to be used for Postgres’ s libiconv got picked up in my application once. Suffice to say, deep shenanigans ensued as my little endian machine suddenly started chugging down big-endian UTF-32 data.) In fact, asking for “UTF-32” does not guarantee there is any relationship between the encoding name you asked for and what is the actual byte representation; despite being a POSIX standard, there are no guarantees about the name \leftrightarrow encoding mapping. There is also no way to control Byte Order Marks, which is hilarious when e.g. you are trying to compile the C Standard using LaTeX and a bad libiconv implementation (thanks Postgres) that inserts them poisons your system installation²⁵.

It is further infuriating that the error handling modes for POSIX can range from “stop doing things and return here to the user can take care of it”, “insert ASCII ? everywhere an error occurs” (glibc does this,

²⁵https://twitter.com/_phantomderp/status/1460695063193870339

and sometimes uses the Unicode Replacement Character when it likes), or even “insert ASCII * everywhere an error occurs” (musl-libc; do not ask me why they chose * instead of the nearly-universally-applied ?). How do you ask for a different behavior? Well, by building an entirely different libiconv module based on a completely different standard library and/or backing implementation of the functionality. Oh, the functionality comes from a library that is part of your core distribution? Well, just figure out the necessary linker and loader flags to get the right functions first. Duh!

Of course. How could I be such a bimbo! Just need to reach into my system and turn into a mad dog frothing at the mouth about encodings to get the behavior that works best for me. I just need to create patches and hold my distribution updates at gunpoint so I can inject the things I need! So simple. So easy!!

In short, libiconv is a great API tainted by a lot of exceedingly poor specification choices, implementation choices, and deep POSIX baggage. Its lack of imagination for a better world and contentment with a broken, lackluster specification is only rivaled by its flaccid, uninspired API usage and its crippling lack of behavioral guarantees. At the very least, GNU libiconv provides a large variety of encodings, but lacks extensibility or any meaningful way to override or control how specific encoding conversions behave, leaving you at the mercy of the system.

In other words, it behaves exactly like every other deeply necessary C API in the world, so no surprises there.

83.7 boost.text

This is perhaps the spiritually most progressive UTF encoding and decoding library that exists. But, while having the ability to perhaps add more encodings to its repertoire, it distinctly refuses to and instead consists only of UTF-based encodings. There is a larger, more rich offering of strictly Unicode functionality (normalization, bidirectional algorithms, word break algorithms, etc.) that the library provides, but we are —perhaps unfortunately —not dealing with APIs outside of conversions for now. Like utf8cpp and simdutf before it, it offers simple free functions for conversions:

```
template <std::input_iterator I, std::sentinel_for<I> S,
^&I     std::output_iterator<uint16_t> O>
        transcode_result<I, O> transcode_to_utf16(
^&I     I first,
^&I     S last,
^&I     O out);
```

It also offers range-based solutions, more fleshed out than utf8cpp’s. These are created from a `boost::text::as_utfN(...)` function call, (where N is 8, 16, 32) and produce iterator/range types for going from the input type (deduced from the pointer (treated as a null-terminated C-string) or from the range’s value type) to the N-deduced output type.

As usual, the criticism of “please do not assume unbounded writes are the only thing I am interested in or need” applies to boost.text’s API here. Thankfully, it does something better than simdutf or utf8cpp: it gives back both the incremented `first` and the incremented `out` iterators. Meaning that if I passed 3 pointers in, I will get 2 updated pointers back, allowing me to know how much was read and how much was written. There is an open question about whether or not one can safely subtract pointers that may have a difference larger than `PTRDIFF_MAX` without invoking undefined behavior, but I have resigned myself that it is more

or less an impossible problem to solve in a C and C++ standards-compliant way for now (modulo relying on not-always-present `uintptr_t`).

`boost.text`'s unfortunate drawback is in its error handling scheme. You are only allowed to insert one (1) character—maybe—through its error handler abstraction, and only when using its iterator-based replacement facilities. During it's `transcode_to_utfN` functions, it actively refuses to do anything other than insert the canonical Unicode replacement character, which means you cannot get the fast iteration functions to stop on bad data. It will just plow right on through, stick the replacement character, and then smile a big smile at you while pretending everything is okay. This can have some pretty gnarly performance implications for folks who want to do things like, say, stop on an error or perform some other kind of error handling, forcing you to use the (less well-performing) iterator APIs.

But this is par-for-the-course for `boost.text`; it was made to be an incredibly opinionated API. I personally think its opinionated approach to everything is not how you get work done when you want to pitch it to save C and C++ from its encoding troubles, and when I sent my mailing list review for it when I still actively participated in Boost I very vocally explained why I think it's the wrong direction. It is, in fact, the one of the bigger reasons I got involved with the C++ Standards Committee SG16: someone was doing something wrong on the internet²⁶ and I couldn't let that pass²⁷:

It's still not a bad library. It has less features and ergonomics versus `simdutf`, but the optimizations Zach Laine put into the encoding conversion layer are slick and at times compete with `simdutf`, when it's working. (More on that later, when we get to the benchmarks.)

83.8 Standard C and C++

I am not even going to deign to consider C++ here. The APIs for doing conversions were so colossally terrible that they were not only deprecated in C++17 (NOTE: not yet removed, but certainly deeply discouraged by existing compiler flags). I myself have suffered an immensely horrible number of bugs trying to use the C++ version from the `<codecvt>`, and users of my `sol2` library have also suffered from the exceedingly poor implementation quality derived from an even worse API that does not even deserve to be mentioned in the same breath as the rest of the APIs here. You can read some of the criticisms:

1. `wstring_convert` sucks²⁸
2. `wstring_convert` constructor repeat many times causes performance degradation²⁹
3. `std::codecvt::out` in LLVM libc++ does not advance in and out pointers³⁰

`<codecvt>` and `std::wstring_convert` are dead and I will never hide how glad I am we put that thing in the trash can.

I'm also less than thrilled about the C Standard API for conversions. There are a number of problems, but I won't regale you all of the problems because I wrote a whole paper about it so I could fix it eventually in C. It did not make C23 because a last minute objection to the structure of the wording handling state ended

²⁶<https://xkcd.com/386/>

²⁷https://thephd.dev/_presentations/unicode/sg16/2018.03.07/2018.03.07%20-%20ThePhD%20-%20a%20rudimentary%20unicode%20abstraction.pdf

²⁸<https://github.com/ThePhD/sol2/issues/571>

²⁹<https://github.com/ThePhD/sol2/issues/326>

³⁰https://github.com/OpenMPT/openmpt/blob/8bc1deb5f01af6b276ae2da32f6bb5873f23df7d/src/mpt/string_transcode/transcode.hpp#L820

WHAT TO DO: A CONVERSATION

ThePhD: So [tzlaine/text](#) it's like Tom's `text_view`, but stapled to `utf8`?

tzlaine: Yes. With quite a few staples.



WHAT TO DO

- De-couple the `text` class (and the `rope` class, too!) from its `utf8` storage mechanism
- Split asunder:
 - into a `text_view` alike class similar to what Tom Honermann's is
 - into `text` class alike to what I used to have (except way better)
 - hammer out free functions and their interfaces (transcoding, etc.)
 - Enjoy the beautiful new C++17/20 in Visual Studio save ICE for hot days 😊



up costing the paper it's ability to make the deadline. Sorry; despite being the project editor I am (A) new to this, (B) extremely exhausted, and (C) not good at this at all, unfortunately!

Nevertheless, the C standard does not support UTF-16 as a wide encoding right now, which violates at least 6 different existing major C platforms today. Even if the wide (`wchar_t`) encoding is UTF-32, the C API is still *fundamentally incapable of representing many of the legacy text encodings it is supposed to handle in the first place*. This has made even steely open source contributors stared slack-jawed at embattled C libraries like glibc, which have no choice but to effectively jettison themselves into nonsense behavior because the C standard provides no proper handling of things. This case, in particular, arises when Big5-HKSCS needs to return two UTF-32 code points (e.g. `U"\U00CA\U+0304"`) for certain input sequences (e.g. `"É"`):

oh wow, even better: glibc goes absolutely fucking apeshit (returns 0 for each `mbrtowc()` after the initial one that eats 2 bytes; herein `wc` modified to write the resulting character)

—наб, July 9, 2022³¹

In fact, implementations can do whatever they want in the face of Big5-HKSCS, since it's outside the Standard's auspices:

...

Florian raised a similar issue in May of 2019 and the general feedback at that time was that BIG5-HKSCS is simply not supported by ISO C. I expect the same answer from POSIX which is harmonized with ISO C in this case.

If BIG5-HKSCS is not supported, then the standard will have nothing to say about which values can be returned after the first or second input bytes are read. ...

—Carlos O' Donnell, March 30, 2020³²

And indeed, the standard cannot handle it. Both because of the assumption that a single `wchar_t` (one UTF-32 code unit, a single `char32_t`) can represent all characters from any character set, and the horrible API design that went into the `mbrtowc/wertomb/etc.` function calls. My paper details much of the pitfalls³³ and I won't review them exhaustively here, but suffice to say everyone who has had their head in the trenches for a long time has conclusively reached the point where we know the original APIs are bunk garbage. I have no intention of rehashing why these utilities are garbage and do not work, and seek only to supplant them and then drive them back into the burning hell whence forth they deigned to sputter out of.

Also, fun fact: the `<uchar.h>` functions which do attempt to do Unicode conversions (but use the execution encoding as the “go between”, so if you do not have a UTF-8 multibyte encoding every encoding is lossy and worthless) are not present on Mac OS. Which is weird, because Mac OS went all-in on UTF-8 encoding conversions as its `char*` encoding in all of its languages, so it could just...make that assumption and ignore all the BSD-like files left lurking in the guts of the OS. But they don't, so instead they just provide ...nothing.

All in all, if the C standard was at least capable —or the C++ standard ever rolled its sleeves up and design something halfway good —we might not be in this mess. But we are, driving the reason for this whole article. Of course, some platforms realized that the C and C++ standards are trash, so they invented their own functions. Like, for example, the Win32 folks.

83.9 Windows API

The Windows API has 2 pretty famous functions for doing conversions: `WideCharToMultiByte` and `MultiByteToWideChar`. They convert from a given code page to UTF-16, and from UTF-16 to a given code page. The signatures from MSDN look as follows:

```
int WideCharToMultiByte(
    UINT                      CodePage,
    DWORD                     dwFlags,
    _In_NLS_string_(cchWideChar)LPCWCH lpWideCharStr,
```

³¹<https://twitter.com/nabijaczleweli/status/1545890979466592257>

³²<https://sourceware.org/pipermail/libc-alpha/2020-March/112300.html>

³³https://thephd.dev/_vendor/future_cxx/papers/C%20-%20Restartable%20and%20Non-Restartable%20Character%20Functions%20for%20Efficient%20Conversions.html

```

int cchWideChar,
LPSTR lpMultiByteStr,
int cbMultiByte,
LPCCH lpDefaultChar,
LPBOOL lpUsedDefaultChar
);

int MultiByteToWideChar(
    UINT CodePage,
    DWORD dwFlags,
    _In_NLS_string_(cbMultiByte)LPCCH lpMultiByteStr,
    int cbMultiByte,
    LPWSTR lpWideCharStr,
    int cchWideChar
);

```

There is no “one by one” API here; just bulk. And, similar to the criticisms levied at simduff, the standard library, and so many other APIs, they only have a single return int that is used as both the error code channel and the return value for the text. (We will ignore that they are using int, which definitely means you cannot be using larger than 4 GB buffers, even on a 64-bit machine, without getting a loop prepped to do the function call multiple times.) I am willing to understand Windows’ s poor design because this API is some literal early-2000s crap. I imagine with all the APIs Windows cranks out regularly, they might have an alternative to this one by now. But if they do, (A) I cannot find such an API, and (B) contacting people who literally work (and worked) on the VC++ runtime have started in no uncertain terms that the C and C++ code to use for these conversions is the [WideCharToMultiByte/MultiByteToWideChar](#) interfaces.

So, well, that’s what we’re using.

This API certainly suffers from its age as well. For example, it does things like assume you would only want to insert 1 replacement character (and that the replacement character can fit neatly in one UTF-16 code unit). This was fixed in more recent versions of windows with the introduction of the [MB_ERR_INVALID_CHARS](#) flag that can be passed to the [dwFlags](#) parameter, where the conversion simply fails if there are invalid characters. Of course, the same problem as simduff manifests but in an even **worse** fashion. Because the error code channel is the same as the “# of written bytes” channel (the return value), returning an error code means you cannot even communicate to the user where you left off in the input, or how much output you have written. If the conversion fails and you want to, say, insert a replacement [u'\xFFFFD'](#) or [u'?'](#) by yourself and skip over the single bit of problematic input, you simply cannot because you have no idea where in the output to put it. You also don’t know where the error has occurred in the input. It’s the old string conversion issue detailed at the start of this article, all over again, and it’s **infuriating**.

83.10 ztd.text

Turns out I have an entire slab of documentation³⁴ you can read about the design, and an entire article explaining part of that design³⁵ out, so I really won't bother explaining ztd.text. It's the API I developed, the API mentioned in the video linked above, and what I've poured way too much of my time into for the sole purpose of saving the C and C++ landscape from its terrible encoding woes. I have people reaching out from different companies already attempting re-implementations of the specification for their platforms, and progress continues to move forward.

It checks every single box in the table's row for the desired feature sets, obviously. If it didn't I would have went back and whipped my API into shape to make sure it did, but I didn't have to because unlike just about every other API in this list it actually paid attention to everything that came before it and absorbed their lessons. I didn't make obvious mistakes or skip over use cases because, as it turns out, listening and learning are really, really powerful tools to prevent rehashing 30 year old discussions.

Wild, isn't it?

83.11 So…What Happens Now?

We listed a few criteria and talked about it, so let's try to make a clear table of what we want out of an API and what each library gives us in terms of conversions. As a reminder, here's the key:

1. **yes** Meets all the necessary criteria of the feature.
2. **no** Does not meet all the necessary criteria of the feature.
3. **:)** Partially meets the necessary criteria with caveats or addendums.

And here's how each of the libraries squares up.

³⁴<https://ztdtext.readthedocs.io/en/latest/design.html>

³⁵<https://thephd.dev/any-encoding-ever-ztd-text-unicode-cpp>

Feature Set ↓ vs. Library →	ICU	libiconv	simdutf	encoding_rs/en	ztd.text
Handles Legacy Encodings	yes	yes	no	yes	yes
Handles UTF Encodings	yes	yes	yes	:)	yes
Bounded and Safe Conversion API	yes	yes	no	yes	yes
Assumed Valid Conversion API	no	no	yes	no	yes
Unbounded Conversion API	no	no	yes	no	yes
Counting API	yes	no	yes	yes	yes
Validation API	yes	no	yes	no	yes
Extensible to (Runtime) User Encodings	no	no	no	no	yes
Bulk Conversions	yes	yes	yes	yes	yes
Single Conversions	yes	no	no	no	yes
Custom Error Handling	yes	:)	:)	yes	yes
Updates Input Range (How Much Read TM)	yes	yes	:)	yes	yes
Updates Output Range (How Much Written TM)	yes	yes	no	yes	yes

Feature Set ↓ vs. Library →	boost.text	utf8cpp	Standard C	Standard C++	Windows API
Handles Legacy Encodings	no	no	:)	:)	yes
Handles UTF Encodings	yes	yes	:)	:)	yes
Bounded and Safe Conversion API	no	no	:)	yes	yes
Assumed Valid Conversion API	yes	yes	no	no	no
Unbounded Conversion API	yes	yes	no	no	yes
Counting API	no	:)	no	no	yes
Validation API	no	:)	no	no	no
Extensible to (Runtime) User Encodings	no	no	no	yes	no
Bulk Conversions	yes	yes	:)	:)	yes
Single Conversions	yes	yes	yes	yes	no
Custom Error Handling	no	yes	yes	yes	no
Updates Input Range (How Much Read TM)	yes	no	yes	yes	no
Updates Output Range (How Much Written TM)	yes	yes	yes	yes	no

Briefly covering the “:)” for each library/API:

1. libiconv: error handling and insertion of replacements is implementation-defined, and the replacements are also implementation-defined, and whether or not it even does it is implementation-defined, and whether or not it’s any good is –you guessed it! —implementation-defined.
2. simduff: it only reports how much output was read one success, and only reports how much input was read if you’re careful, making inserting custom handling a lot harder than is necessary.
3. encoding_rs: cannot handle UTF-32 from C or C++ (but gets it for free in Rust because you can convert to Rust char, which is a Unicode Scalar Value).
4. boost.text: this one has an “**no**” for custom error handling, despite enabling it for its ranges, because its bulk transcoding functions refuse you the opportunity or chance to do that and they do not provide functions to allow you to change it.
5. utf8cpp: it does not provide counting and validation APIs for all UTF functions, so even if restricted to purely UTF functions validation and counting must be done by the end-user, or through using iterators with a slower API.

6. Standard C: it's trash.
7. Standard C++: provides next-to-nothing of its own that is not sourced from C, and when it does it somehow makes it worse. Also trash.
8. Windows API: it does not handle UTF-32. From the documentation of its Code Page identifiers³⁶ (emphasis mine): UTF-32 continues to be for losers, apparently!

This is the full spread. Every marker should be explained above; if something is missing, do let me know, because I am going to routinely reference this table as the Definitive™ Feature List for all of these libraries from now until I die and also in important articles and journals. I spent way too much time investigating these APIs, suffering through their horrible builds, and knick-knack-patty-whacking these APIs and benchmarks and investigations together. I most certainly never want to touch libiconv again, and even though I'm tired as hell I've already put "remake ztd.text in Rust so I can have an UTF-32 conversion as part of a Rust text library, For God's Sake" on my list of things to do. (Or someone else will get to do it before I do, which would be grreeat.)

But...Where's Your C API?

Right. I said I was going to use all of this to see if we can make an API in C that matches the power of my C++ one, and learns all the necessary lessons from all the C and C++ APIs that litter the text encoding battlefield. A C API for working with text that covers all of the use cases and basis that already exist in the industry. Which is exactly what I did when I created ztd.cuneicode, a powerful C library that allows runtime extension and uncompromised speed while absorbing the lessons of Stepanov's STL, iconv's interface, and libogonek/ztd.text's state handling apparatus. The time has come to explain the ultimate C encoding API to you

83.12 ... Part 2

Sorry! It turns out this article has quite literally surpassed 10,000 words and quite frankly there's still a LOT to talk about. The next one might another 10,000 word banger (and I sincerely do not want it to be because then that means I will be writing far past New Years and into 2023). So, the actual design of the C library, its benefits, and more, will all come later. But I won't just leave you empty-handed! In fact, here's a little teaser...

Nyeheheh, such beautiful graphs...!

See you soon :).

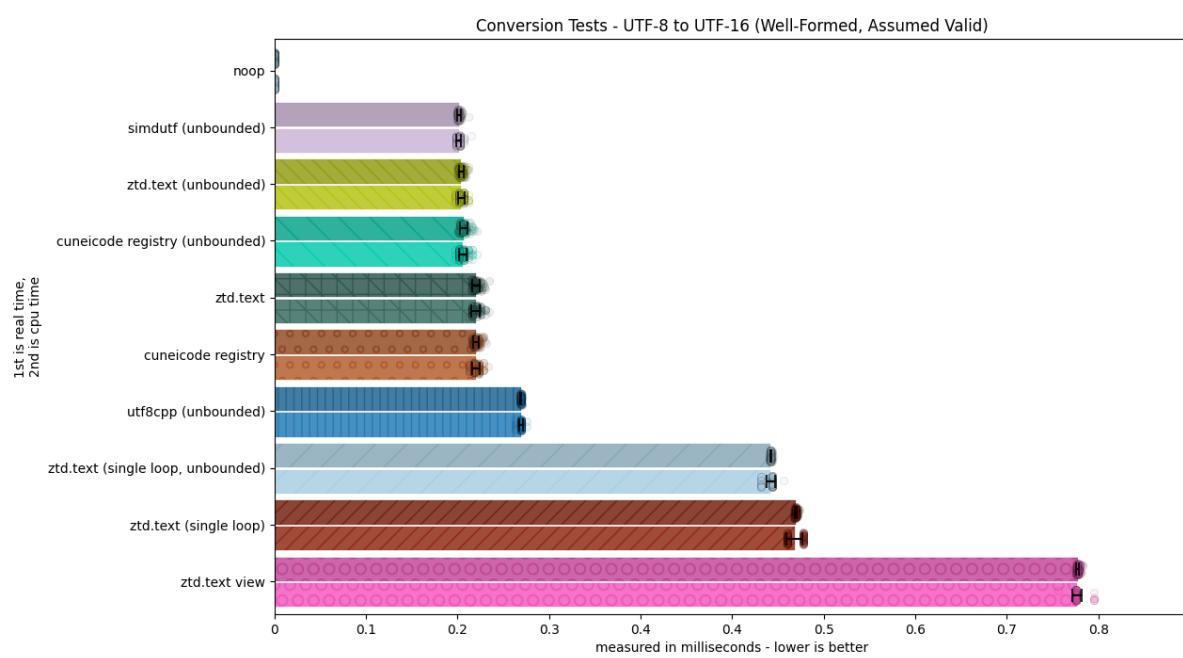
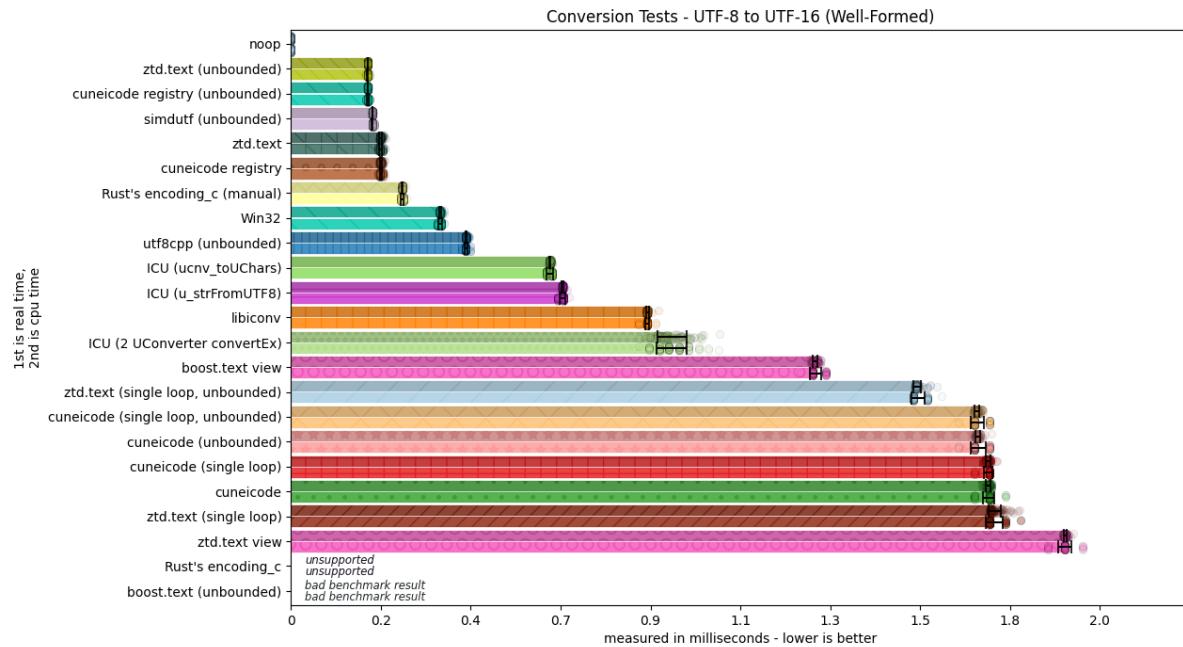
1. Thumbnail / Title Feature art done by Nemo (@WusdisWusdat)³⁷ (There's a whole comic, but it's being Saved™ for a whole new blog post!)
2. "a" Emote art done by Framebuffer (@framebuffer)³⁸
3. Exhausted Phone art done by Cynthia (@PixieCatSupreme)³⁹
4. Concerned Emote art done by Queenie B

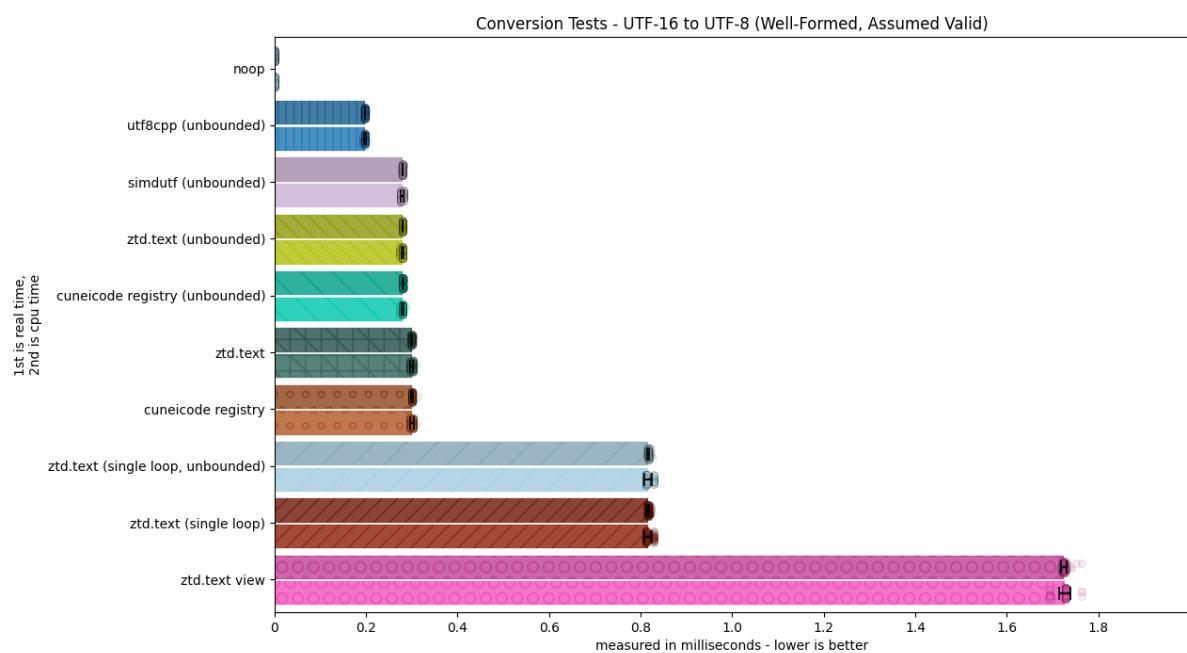
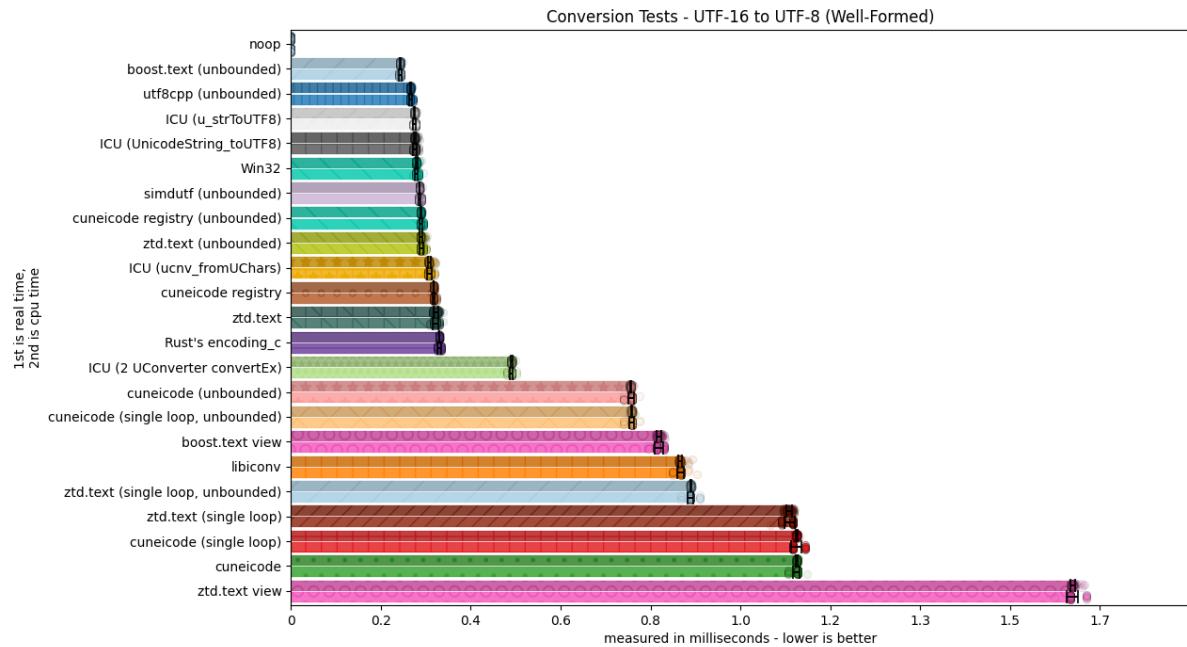
³⁶<https://learn.microsoft.com/en-us/windows/win32/intl/code-page-identifiers>

³⁷<https://twitter.com/WusdisWusdat>

³⁸<https://twitter.com/framebuffer>

³⁹<https://twitter.com/PixieCatSupreme>





The Links of All Articles

- [1] <https://tinyurl.com/overload-resolution>
- [2] <https://medium.com/@simontoth/move-semantics-and-rvalue-references-modern-c-fundamentals-cbbe38760c05>
- [3] <https://thephd.dev/to-save-c-we-must-save-abi-fixing-c-function-abi>
- [4] <https://mariusbancila.ro/blog/2022/01/01/the-evolution-of-functions-in-modern-cpp/>
- [5] <https://pvs-studio.com/en/blog/posts/cpp/0909/>
- [6] <https://quuxplusone.github.io/blog/2022/06/03/aggregate-parens-init-considered-kinda-bad/>
- [7] <https://www.cppstories.com/2015/02/non-static-data-members-initialization/>
- [8] <https://www.codingwiththomas.com/blog/c-static-dynamic-polymorphism-crtp-and-c20s-concepts>
- [9] <https://www.cppstories.com/2022/structured-bindings/>
- [10] <https://felipe.rs/2021/09/19/std-optional-and-non-pod-types-in-cpp/>
- [11] <https://www.foonathan.net/2021/07/concepts-structural-nominal/>
- [12] <https://quuxplusone.github.io/blog/2023/04/08/most-ctors-should-be-explicit/>
- [13] <https://tinyurl.com/overview-of-cpp23-features>
- [14] <https://mariusbancila.ro/blog/2022/01/17/three-cpp23-features-for-common-use/>
- [15] <https://mariusbancila.ro/blog/2022/11/08/three-new-utility-functions-in-cpp23/>
- [16] <https://devblogs.microsoft.com/cppblog/cpp23-deducing-this/>
- [17] <https://mariusbancila.ro/blog/2022/08/17/using-the-cpp23-expected-type/>
- [18] <https://www.sandordargo.com/blog/2022/06/01/cpp23-if-consteval>
- [19] <https://www.sandordargo.com/blog/2022/06/15/cpp23-narrowing-contextual-conversions-to-bool>
- [20] https://www.sandordargo.com/blog/2022/07/20/6-features-improving-string-string_view-in-cpp23
- [21] <https://www.sandordargo.com/blog/2022/09/07/prepocessive-directive-changes-in-cpp23>
- [22] <https://www.sandordargo.com/blog/2022/09/21/cpp23-stacktrace-library>

- [23] https://www.sandordargo.com/blog/2022/10/05/cpp23-flat_map
- [24] <https://tinyurl.com/cpp23-lambdas-change>
- [25] <https://www.sandordargo.com/blog/2022/12/14/cpp23-attributes>
- [26] <https://github.com/kokkos/mdspan/wiki/A-Gentle-Introduction-to-mdspan>
- [27] <https://www.elbeno.com/blog/?p=1696>
- [28] <https://devblogs.microsoft.com/cppblog/cpp23s-optional-and-expected/>
- [29] <https://www.sandordargo.com/blog/2023/02/15/evolution-ofEnums>
- [30] <https://tinyurl.com/introduction-of-cpp-reflection>
- [31] <https://tinyurl.com/dynamic-and-static-reflection>
- [32] <https://tinyurl.com/circle-lang-reflection>
- [33] <https://tinyurl.com/static-reflection>
- [34] <https://www.accu.org/journals/overload/30/168/teodorescu/>
- [35] <https://blog.m-ou.se/rust-cpp-concurrency/>
- [36] <https://ggulgulia.medium.com/c-20-concurrency-part-1-synchronized-output-stream-59532e85cde8>
- [37] https://vector-of-bool.github.io/2021/12/30/co_resource.html
- [38] <https://lewissbaker.github.io/2022/08/27/understanding-the-compiler-transform>
- [39] <https://ladnir.github.io/blog/2022/01/24/macoro.html>
- [40] <https://clang.llvm.org/docs/DebuggingCoroutines.html>
- [41] <https://pabloariasal.github.io/2022/11/12/couring-1/>
- [42] <https://johnnysswlab.com/instruction-level-parallelism-in-practice-speeding-up-memory-bound-programs-with-low-ilp/>
- [43] <https://tinyurl.com/ways-initialize-string-member>
- [44] <https://blog.feabhas.com/2022/11/using-final-in-c-to-improve-performance/>
- [45] <https://www.foonathan.net/2022/01/compile-time-codegen/>
- [46] <https://devblogs.microsoft.com/cppblog/improving-the-state-of-debug-performance-in-c/>
- [47] <https://www.cppstories.com/2018/03/ifconstexpr/>
- [48] <https://johnnysswlab.com/the-memory-subsystem-from-the-viewpoint-of-software-how-memory-subsystem-effects-software-performance-1-2/>
- [49] https://vittorioromeo.info/index/blog/debug_performance_cpp.html
- [50] <http://www.modernescpp.com/index.php/using-requires-expression-in-c-20-as-a-standalone-feature>

- [51] https://johnysswlab.com/what-is-faster-vec-emplace_backx-or-vecx/
- [52] <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2723r0.html>
- [53] <https://vittorioromeo.info/index/blog/wordexpr.html>
- [54] <https://www.cppstories.com/2022/const-options-cpp20/>
- [55] <https://www.cppstories.com/2022/ranges-alg-part-one/>
- [56] <https://devblogs.microsoft.com/cppblog/cpp23s-new-fold-algorithms/>
- [57] <https://jk-jeon.github.io/posts/2022/02/jeaiii-algorithm/>
- [58] <https://brevzin.github.io/c++/2021/11/21/conditional-members/>
- [59] <https://bannalia.blogspot.com/2022/11/inside-boostunorderedflatmap.html>
- [60] <https://en.algorithmica.org/hpc/data-structures/s-tree/>
- [61] <https://www.foonathan.net/2022/05/recursivariant-box/>
- [62] <https://blog.feabhas.com/2022/02/working-with-strings-in-embedded-c/>
- [63] <https://www.lukas-barth.net/blog/checking-if-specialized/>
- [64] <https://www.fluentcpp.com/2022/05/16/how-to-store-an-lvalue-or-an-rvalue-in-the-same-object/>
- [65] <https://indiegamedev.net/2022/03/28/automatic-serialization-in-cpp-for-game-engines/>
- [66] <https://tinyurl.com/constexpr-string-to-runtime>
- [67] <https://tinyurl.com/print-variables-type>
- [68] <https://eversinc33.github.io/posts/avoiding-direct-syscall-instructions/>
- [69] <https://www.modernescpp.com/index.php/avoiding-temporaries-with-expression-templates>
- [70] <https://embeddeduse.com/2013/08/25/casting-a-negative-float-to-an-unsigned-int/>
- [71] <https://randomascii.wordpress.com/2022/12/14/compiler-tricks-to-avoid-abi-induced-crashes/>
- [72] <https://www.cppstories.com/2022/five-topics-data-members-cpp20/>
- [73] <https://devblogs.microsoft.com/oldnewthing/20220207-00/?p=106223>
- [74] <https://www.foonathan.net/2022/09/new-integer-types/>
- [75] <https://devblogs.microsoft.com/cppblog/proxy-runtime-polymorphism-made-easier-than-ever/>
- [76] <https://thephd.dev/c23-is-coming-here-is-what-is-on-the-menu>
- [77] <https://herbsutter.com/2022/12/31/cpp2-and-cppfront-year-end-mini-update/>
- [78] <https://www.fluentcpp.com/2022/04/26/copy-paste-developments/>
- [79] <https://www.foonathan.net/2022/07/carbon-calling-convention/>

- [80] <https://www.cppstories.com/2017/03/on-toggle-parameters/>
- [81] <https://www.foonathan.net/2022/08/malloc-interface/#content>
- [82] <https://developers.redhat.com/articles/2022/06/02/use-compiler-flags-stack-protection-gcc-and-clang#>
- [83] <https://thephd.dev/the-c-c++-rust-string-text-encoding-api-landscape>